AUTOMATIC SYNTHESIS AND VERIFICATION OF GATE-LEVEL TIMED CIRCUITS

Chris J. Myers
Tomas G. Rokicki
Teresa H.-Y. Meng

Technical Report: CSL-TR-94-652

December, 1994

This research was supported by an NSF fellowship and a research grant by ARPA.

AUTOMATIC SYNTHESIS AND VERFICATION OF GATE-LEVEL TIMED CIRCUITS

Chris J. Myers, Tomas G. Rokicki, and Teresa H.-Y. Meng

Technical Report: CSL-TR-94-652

December, 1994

Computer Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, CA 94305-4055
pubs@shasta.stanford.edu

Abstract

This paper presents a CAD system for the automatic synthesis and verification of *gate-level timed circuits.* Timed circuits are a class of asynchronous circuits that incorporate explicit timing information in the specification which is used throughout the synthesis procedure to optimize the design. The system accepts a textual specification capable of specifying general circuit behavior and timing requirements. This specification is automatically transformed to a graphical representation that can be analyzed using an exact and efficient timing analysis algorithm to find the reachable state space. From this state space, our synthesis procedure derives a timed circuit that is hazard-free using only basic gates to facilitate the mapping to semi-custom components, such as standard-cells and gate-arrays. The resulting gate-level timed circuit implementations are up to 40 percent smaller and 50 percent faster than those produced using other asynchronous design methodologies. We also demonstrate that our timed designs can be smaller and faster than their synchronous counterparts. We have also applied our timing analysis algorithm to verify efficiently not only our synthesized circuits but also a wide collection of large, highly concurrent timed circuits that could not previously be verified using traditional techniques.

Key Words and Phrases:

Computer-aided design, asynchronous circuit synthesis, orbital nets, partial order timing, state graphs, hazard-free logic synthesis, gate-level timed circuits, and timed circuit verification.

Contents

1 Introduction

In recent years, there has been a resurgence of interest in the design of *asynchronous circuits* due to their ability to eliminate clock skew problems, achieve average case performance, adapt to processing and environmental variations, and provide component modularity. Asynchronous circuits can also lower system power requirements by reducing synchronization power, automatically powering down unused components, removing spurious transitions, and easily adjusting to a dynamic power supply. While asynchronous designs have long been used in interface circuits, they are now being considered for the design of low-power embedded controllers and portable devices due to their low-power advantages.

Traditional academic asynchronous design methodologies use unbounded delay assumptions, resulting in circuits that are verifiably correct, but sacrifice timing for simplicity, leading to unnecessarily conservative designs. In industry, however, timing is critical to reduce both chip area and circuit delay. Due to the lack of formal methods to handle timing information correctly, circuits with timing constraints usually require extensive simulation to gain confidence in the design. Our research bridges this gap by introducing *timed circuits* in which explicit timing information is incorporated into the specification and utilized throughout the design procedure to optimize the implementation. Timed circuits can be significantly smaller and faster than those produced using traditional methods, and they are more reliable than those produced using adhoc methods. The specification of timing constraints also facilitates a natural interaction between synchronous and asynchronous circuits.

Timing considerations, however, often introduce an additional exponential factor of complexity into the design procedure. As a result, timing analysis has hitherto either been avoided [1, 2, 3], simplified [4, 5], or considered only after synthesis [6]. In this paper, we develop an exact and efficient timing analysis algorithm, and apply it to the automatic synthesis and verification of gate-level timed circuits.

In our previous work, an efficient timing analysis algorithm is developed and applied to incorporate timing considerations into the synthesis of timed circuits [7]. We verified that our timed circuit implementations are hazard-free using Burch's timed circuit verifier [8]. This work, however, is not without its limitations. First, since the timing analysis is limited to only deterministic specifications, our synthesis procedure could only be applied to a limited class of circuits. Second, our timed circuit implementations require complex-gates, making it difficult to use semi-custom components, such as standard-cells and gate-arrays, which are becoming increasingly important to improve time-to-market. Third, using the discrete-time verification approach employed by Burch's verifier is limited in its applicability since the number of discrete-time states grows exponentially with respect to the number of concurrent events.

In this paper, we present a more general and widely applicable procedure for the synthesis and verification of timed circuits. First, we expand the class of specifications to allow conditional operation, or choice, by applying automatic transformations to the specification to obtain a representation which can be analyzed by an exact and efficient timing analysis algorithm. Second, we facilitate the mapping of our implementations to semi-custom components by adding constraints to our synthesis procedure, thereby, producing hazard-free timed circuits using only basic gates such as AND gates, OR gates, and C-elements. Our synthesis procedure has been fully automated in a CAD tool and applied to several examples, resulting in gate-level timed circuit implementations which are up to 40 percent smaller and 50 percent faster than designs using other methodologies. After synthesis, using the same timing analysis procedure, we then verify if the synthesized timed circuit implementation back-annotated with delays from a given cell-library satisfies its timed specification. Our verification procedure is shown to be able to rapidly verify larger, more concurrent timed circuits than could previously be verified using traditional techniques.

In section 2, we describe the initial textual specification language and the graphical representation translated to for timing analysis. Section 3 describes our timing analysis algorithm and how it is used to explore the timed state space. Section 4 explains the synthesis procedure, and section 5 describes the verification procedure. Section 6 gives our conclusions.

2 Timed specifications

Many approaches could be taken to specify timed circuits including using languages such as *communicating sequential processes* (CSP) [1] or graphs such as *signal transition graphs* (STG) [2]. Graphs are conducive to automated timing analysis and synthesis algorithms, but they are cumbersome for specifying a large system. Languages, however, allow large designs to be specified clearly and concisely. For these reasons, we chose to use a language, *timed handshaking expansions* (THSE), as the initial specification which is then compiled to a graphical representation, *an orbital net* [9], for timing analysis. This section describes our specification language, our graphical representation, and the procedure used to compile from a specification to a representation which can be efficiently analyzed.

2.1 Timed handshaking expansions

THSE are based on Martin's *handshaking expansions* to which timing has been added and are easily derivable from a CSP specification augmented with timing information using a method similar to Martin's [1]. A specification using THSE is composed of two parts: a set of signal declarations and a set of *processes* executing in parallel. Each declaration consists of a type (either *input* or *output),* a signal name, an initial value (either *true* or *false),* and delays associated with transitions on the signal. A delay is given in the form: $\langle l_r$, u,; l_f , u_f) where l_r and u,. are the lower and upper bounds on a rising transition and l_f and u_f are the lower and upper bounds on a falling transition. If the fall times are not specified, they are assumed to be equal to the rise times. Each process is a set of commands repeated forever (denoted $*$[C]). These commands can be executed either in sequence (denoted C_1 ; C_2) or in parallel (denoted $C_1 \,||\, C_2$). Each basic command is either *an event* or a *wait.* An event specifies when the process *causes* the occurrence of either a rising transition (denoted s $\uparrow$) or a falling transition (denoted s $\downarrow$) on a signal s. A wait, on the other hand, specifies when the process must *stall* for a certain set of events (denoted *[event-list*]) caused by some other process(es). The events in the set can be composed *conjunctively* (denoted e_1 A . . . A e,) to specify that the process waits until it has seen all the events in the set. The events can also be composed *disjunctively* (denoted e_1V . . . V e,) to specify that the process waits until it has seen exactly one mutually exclusive event in the set. Since the semantics of the orbital net representation used in our timing analysis is event-based, the semantics of our waits are based on events on signals rather than on values of signals. This change is made explicit in the language by using lists of events rather than predicates on signal values in a wait command.

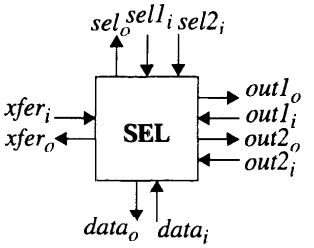
Within a process, a choice of behavior made by the environment may be specified. Choice is represented with a set of *guarded commands* (denoted [$G_1 \rightarrow C_1 \,|\, \ldots \,|\, G_n \rightarrow C_n$]). Each guarded command is composed of two parts: a guard G_i, which is either an event or a wait, and a set of commands C_i. If the guards in a set of guarded commands are waits, the commands associated with the first wait to be satisfied are executed. If the guards are events, one event is nondeterministically chosen resulting in the occurrence of the event followed by the execution of the commands that it guards. Since we allow choice but do not allow arbitration, an event as a guard must be on a signal of type *input.* If an arbiter is needed, it can be added as a special environment process. A guarded command may also loop (denoted $G_i \rightarrow C_i$; $*$) [10]. If a guarded command that loops is selected, then after the set of commands is executed, control is returned to the beginning of the guarded command. This looping continues until a guarded command that does not loop is selected.

The CSP specification for a port selector (SEL) is shown in Figure l(a). The CSP specification dictates the ordering of communications on channels, but many different THSE using the signal wires shown in Figure l(b) could implement the communications. Part of one possible THSE is shown in Figure 2 including a few declarations, the process for the SEL being designed, and the environment process which makes the choice of which output port to use. The basic operation of the SEL is as follows. First, the SEL waits until it gets a request for a data transfer (i.e., $xfer_i \uparrow$), then concurrently issues requests for the selection of an output

port (i.e., $sel_o \uparrow$) and for the data to be transferred (i.e., *data,* $\uparrow$). After the SEL receives the port selection (i.e., *sell;* $\uparrow$ or $sel2_i \uparrow$) and acknowledgment that the data is ready (i.e., *data;* $\uparrow$), it initiates the transfer of the data onto the selected output port (i.e., $out1_o \uparrow$ or $out2_o \uparrow$). After the SEL receives acknowledgment that the selected port received the data (i.e., $out1_i \uparrow$ or $out2_i \uparrow$), it may acknowledge the completion of the data transfer (i.e., $xfer_o \uparrow$). The rest of the signals deal with resetting the four-phase handshake used to implement the communication.

```
*[ [ xfer → ( data || sel?(sell, sel2) );
   [ sell → out1; xfer
   I se12 → out2; xfer
11
```

(a)



(b)

Figure 1: (a) CSP specification and (b) block diagram for a port selector (SEL).

input $sell_i = \{false, (40,260;\ 2,40\rangle\}$;
input $sel2_i = \{false, (40,260;\ 2,40\rangle\}$;
output $sel_o = \{false, \langle 0,20\rangle\}$;
etc.
$* [\ [xfer_i \uparrow]; (([data; \downarrow]; data, \uparrow) \,||\, ([sell_i \downarrow \vee sel2_i \downarrow]; sel_o \uparrow)); [data_i \uparrow];$
$\quad I\ [sell; \uparrow \text{ A } out1_i \downarrow] \rightarrow outl, \uparrow; sel_o \downarrow; [out1_i \uparrow]; (xfer_o \uparrow || data, \downarrow); outl. \downarrow; [xfer_i \downarrow]; xfer_o \downarrow$
$\quad |\ [sel2_i \uparrow A\ out2_i \downarrow] \rightarrow out2_o \uparrow; sel_o \downarrow; [out2_i \uparrow]; (xfer_o \uparrow || data, \downarrow); out2_o \downarrow; [xfer_i \downarrow]; xfer_o \downarrow$
11
||
$* [\ [sel_o \uparrow]; [\ sell_i \uparrow \rightarrow [sel_o \downarrow]; sell_i \downarrow |\ sel2_i \uparrow \rightarrow [sel_o \downarrow]; sel2_i \downarrow\ ]\ ]$
||etc.

Figure 2: Part of the THSE specification for the SEL.

2.2 Orbital nets

The THSE is translated to an orbital net representation. An orbital net is essentially a labeled safe Petri net extended with automatic net constructions and syntactic shorthands for *composition* and *receptiveness* [9]. The net constructions allow us to have relatively straightforward operational semantics, while the syntactic shorthands allow us to compose the nets without an exponential blowup in net size. These features are described in detail in [9]. Orbital nets also include both *behavior-* and *constraint* timing requirements as well as *simultaneous* actions (i.e., transitions labeled with sets of actions). These allow us to easily mix behavior and environmental requirements even at the gate model level. These last two features are described in detail in the following subsections.

An orbital net is modeled by the tuple $\langle A, P, T, F, M_0, R, L\rangle$ where A is the set of atomic actions, P is the set of places, T is the set of transitions, $F \subseteq (P \times T) \cup (T \times P)$ is the set of edges, $M_0 \subseteq P$ is the initial marking, R is an assignment of timing requirements to places, and L is a function which labels transitions with sets of simultaneous actions. A *marking* is a subset of the places. For a place $p \in P$, the *preset* of p (denoted $\bullet p$) is the set of transitions connected to p (i.e., $\bullet p = \{t \in T \mid (t,p) \in F\}$), and the *postset* of p (denoted $p\bullet$)

is the set of transitions to which p is connected (i.e., $p\bullet = \{t \in T \mid (p,t) \in F\}$). For a transition $t \in T$, the presets and postsets are similarly defined (i.e., $\bullet t = \{p \in P \mid (p,t) \in F\}$ and $t\bullet = \{p \in P \mid (t,p) \in F\}$).

2.2.1 Timing requirements

Timing in an orbital net is associated with a place as a timing requirement consisting of a lower bound, an upper bound, and a type (denoted $(I, u\rangle type$). The lower bound is a nonnegative integer and the upper bound is an integer greater than or equal to the lower bound, or ∞. Since real values can be expressed as rationals within any required accuracy, restricting the bounds of timing requirements to be integers does not decrease the expressiveness of orbital nets. Since there are only a finite number of timing parameters, if any are rational, we can multiply all of them by the least common denominator.

There are two types of timing requirements: *behavior (b)* and *constraint(c)*. Behavior timing requirements are used to specify guaranteed timing behavior. Constraint timing requirements, on the other hand, are used to specify desired timing behavior, and they do not affect the actual timing behavior. If the timing requirement on a place is omitted, it is assumed to be $(0, \infty\rangle c$.

Consider a D-type flip-flop (FF) pictured in Figure 3(a). The timing requirements for the FF are depicted using a timing diagram in Figure 3(b) and using an orbital net in Figure *3(c)*. This *FF* has a *setup time* of 5 time units which is represented with a constraint timing requirement from the rising transition on the input D to the rising transition on the clock φ. Similarly, a *hold time* of 5 time units is represented with a constraint timing requirement from the rising transition on the clock φ to the falling transition on the input *D*. Note that these are requirements that the environment must satisfy, and the FF cannot guarantee this behavior. The delay of the FF is represented as a behavior timing requirement from the rising transition of the clock φ to the rising transition on the output Q. This requirement says that the FF circuit will generate Q $\uparrow$ between 5 and 8 time units after $\varphi\uparrow$.

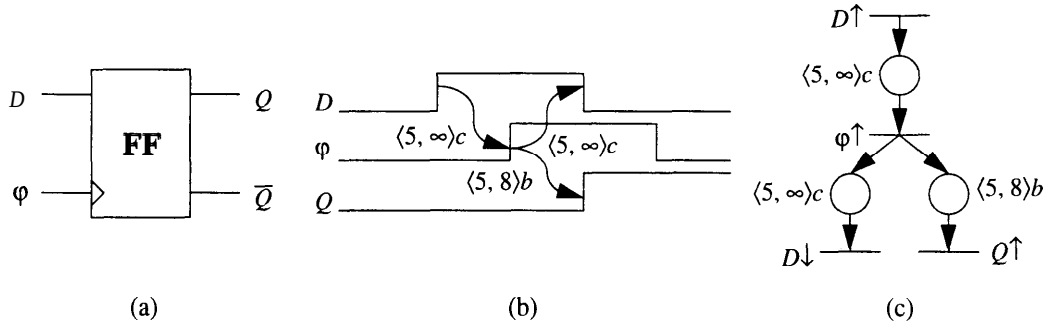


Figure 3: (a)A D-type flip-flop, (b) its timing requirements represented using a timing diagram, and (c) its timing requirements represented using an orbital net.

-When there is a single behavior place p in the preset of a transition, regardless of the interpretation, the time of occurrence of a transition in the postset of p (denoted $t(p\bullet)$) is always greater than the time of occurrence of any transition in the preset of the place (denoted $t(\bullet p)$) by at least the lower bound of the timing requirement on p, and it is always less then the upper bound. If, on the other hand, there are multiple behavior places in the preset of a transition, there are four different ways the specified behavior can be interpreted [5]. The first, or *type 1*, says that for all behavior places p, $t(p\bullet) - t(\bullet p)$ must exceed the lower bound but must not exceed the upper bound (this is the type used by our constraint places). If no possible timing behavior satisfy these constraints, the specification is inconsistent. The second, or *type* 2, says that for all behavior places p, $t(p\bullet) - t(\bullet p)$ must exceed the lower bound and for at least one behavior place, $t(p\bullet) - t(\bullet p)$ must not exceed the upper bound. This is the type usually associated with circuit behavior, so it is the type we associate with our behavior places. *Types* 3 and 4 are duals in which only a single lower bound needs to be

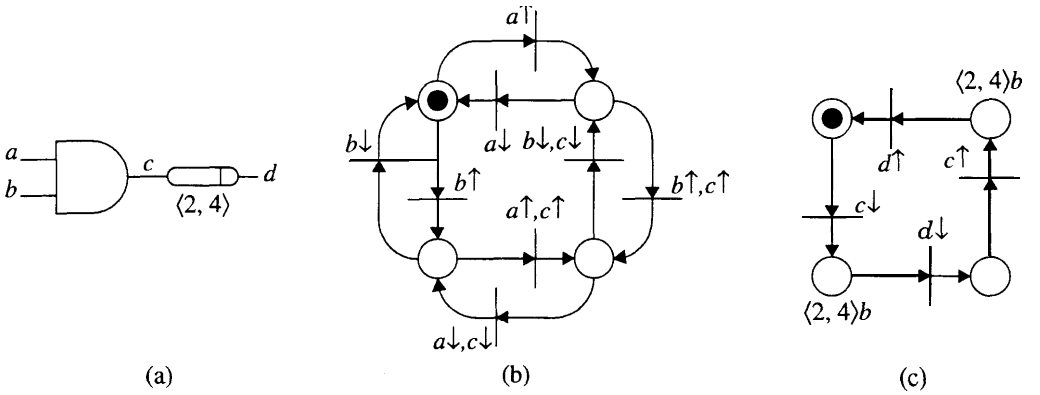


Figure 4: (a) AND gate with inputs a and b, and output d; (b) orbital net for functional behavior; (c) delay buffer with input c, output d, and delay of $\langle 4, 10 \rangle$.

reached (i.e., an OR relationship). These two types are not considered as they do not correspond with the conjunctive nature of the Petri-net model.

Our timing analysis algorithm relies on the fact that each behavior place represents a single nondeterministic choice of delay that cannot be affected by other behavior places. When there are multiple behavior places in the preset of a transition, the type 2 semantics allow the delay between the transition in the preset and postset of a behavior place to exceed its upper bound if the transition in the postset is being constrained by another behavior place. Therefore, our algorithm requires specifications to include at most a single behavior place in the preset of each transition. Fortunately, our original orbital net specification can always be transformed, as described later, into one which satisfies the single behavior place requirement.

2.2.2 Simultaneous actions

For a large class of speed-independent and delay-insensitive designs, any hazard is potentially fatal [11], so simple delay models that are easy to integrate into gate models suffice. With the more complex delay models required for modeling real-time circuit delay, such integration is no longer easy or straightforward. Labeling each transition in an orbital net with a (possibly empty) set of simultaneous actions remedies this difficulty by allowing the function of a gate to be modeled separate from its delay behavior without a significant blowup in the state space size.

Consider, for example, an AND gate with a delay of 4 to 10 time units. Under the *output delay model*, the gate is modeled with an instantaneous function block followed by a delay element as shown in Figure 4(a). The orbital net corresponding to the functional behavior of the AND gate is given in Figure 4(b). In this net, there are four places corresponding to the four states of the two input signals a and b, and the value of c in each place tracks exactly the AND of the signals a and b. The orbital net corresponding to a simple delay element is shown in Figure 4(c). The behavior place labeled (4,10) indicates that an output will occur between 4 and 10 time units after the preceding input occurs; no behavior violating this requirement will be generated by the net. The constraint places do not constrain the behavior of the net, but if another input event occurs before the preceding output event then the environment violates the specification. Composition of these nets gives a AND gate operating under the output delay model. In a similar manner, an AND gate operating under the *input delay model* could also be obtained.

The delay model shown in Figure 4(c) is relatively simple, and it suffices for many types of circuits. More complex delay models can and have been constructed, modeling more accurately the behavior of a gate under hazard conditions; for these, the separation of gate models into combinational function and delay behavior is essential [9].

2.2.3 Operational semantics

The behavior specified by an orbital net that satisfies the single behavior place requirement is defined with an operational semantics composed of two types of operations: advancement of time and firing of transitions. In an orbital net, an *untimed state* is a marking of the net. A *timed state* is an untimed state with a time-valued clock clk_i associated with each marked place. Each clock advances with time and denotes how long the place has been marked. Time is advanced by uniformly increasing these clocks by an amount τ which is less than or equal to *max-advance* for a given marking. The function *max-advance* is defined as the minimum difference over all marked behavior places between the upper bound of the timing requirement on the place and its clock, or ∞ if there are no marked behavior places. This upper limit on time advancement maintains the clocks for all behavior places below the maximum allowed by their range.

In an orbital net, a transition is *untimed-enabled* if all places in its preset are marked. A transition is *timed-enabled* when it is untimed-enabled and if there is a behavior place in its preset, this place's clock is greater than the lower bound of the timing requirement on the place. Any timed-enabled transition can be fired instantaneously, and any number of transitions can be fired without time advancing. A transition is fired by removing the marking in the places in its preset and discarding the clocks. The places in the postset of the fired transition are then marked, and all newly marked places are assigned a clock initialized to zero.

Before firing a transition, however, the constraint places in the entire net must be checked, and if any are marked with clk_i > u, this firing is marked as a failure. Also, the clocks corresponding to a marking that is removed from a constraint place must be checked, and if $clk_i < l$, this firing is also marked as a failure. Finally, after the firing of a transition, every marked behavior place must have a transition in its postset that is untimed-enabled in the new state; if this condition is not satisfied, this firing is a failure. This requirement ensures that every marked behavior place can fire in all states in which its timing conditions are met, and thus the value of its clock when it fires cannot be controlled by external state. If a failure is detected during synthesis, the specification is inconsistent and must be modified before a implementation can be obtained. If a failure is detected during verification, the timed circuit violates its specification.

These semantics define the set of legal *timed firing sequences P,* which is a sequence of pairs of transition firings and time values. For simplicity, the time value represents a non-negative duration since the previous pair. Executing a timed firing sequence a on an orbital net results in the timed state *fire(a).* The set P is defined recursively. The empty sequence ε is in P. For every firing sequence a in P and for every value of τ such that $\tau \leq max\text{-}advance(fire(\alpha))$, then $\alpha(\phi, \tau)$ is in P, where ϕ represents an 'empty' firing. In addition, if a transition t is timed-enabled in *fire(a),* then $\alpha(t, 0)$ is also in P. The reachable state space is the range of the function *fire* over P.

2.3 Translation from a timed handshaking expansion to an orbital net

The procedure to translate a THSE into an orbital net first transforms each concurrent process individually, then composes the resulting orbital nets to obtain the complete orbital net representation. While this process can be formalized, only an intuitive description is given here as it is beyond the scope of this paper.

The procedure to transform a process to an orbital net is initially described without choice. First, for each signal in the specification a rising and falling transition on the signal are added to the set of atomic actions. Next, a transition is added to the net for each occurrence of an action in either an event or a wait, and the transition is labeled with the corresponding action. For each event in the specification, the procedure adds a behavior place and corresponding edges to the corresponding transition from each action in an event or wait that directly precedes it with a timing requirement taken from the declarations. The procedure also adds a constraint place with timing requirement $(0, \infty\rangle c$ and corresponding edges to the preset for any other event which is separated only by waits. To handle the outer loop, the procedure considers the last events as preceding the first events with the difference that each place that is added is initially marked.

To address choice, care must be taken to determine whether a new place is needed or if a previously added place is to be shared. For example, for the selection process from the SEL, the events sell; ↑ and $sel2_i$ ↑ must share the place in the postset of the event sel_o ↑. The complete orbital net for the selection process is shown in Figure 5(a).

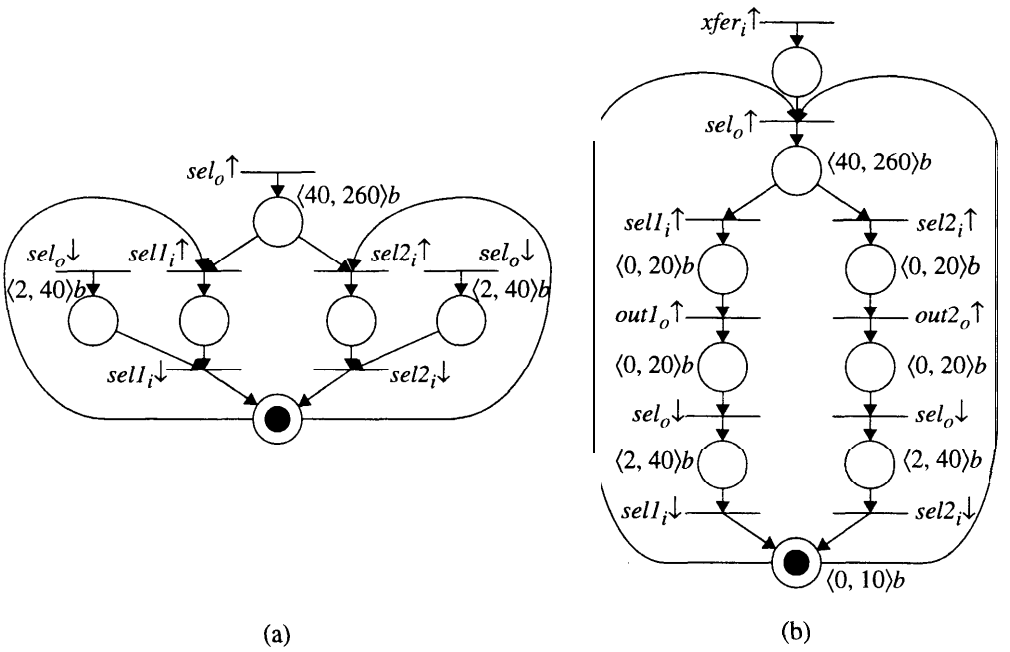


Figure 5: (a) Orbital net for the selection process from the SEL; (b) part of the orbital net after composition with the SEL process.

When this net is composed with the orbital net for the SEL process, different occurrences of actions in one process are matched with their corresponding occurrences in the other process by keeping track of their order. Part of the orbital net after composition with the SEL process is shown in Figure 5(b). Note that as an optimization a simple analysis of the graph determines that all the constraint places in Figure 5(a) can be removed since there are paths through behavior places that make them redundant.

2.4 Satisfying the single behavior place requirement

Next, the orbital net is transformed to one which satisfies the single behavior place requirement. To accomplish this, consider a fragment of an orbital net that has two behavior places in the preset of a transition shown in Figure 6(a). The desired timing behavior can be depicted graphically as shown in Figure 6(b). This net can be transformed to the one shown in Figure 7(a) which satisfies the single behavior place requirement. Basically, the idea behind this net transformation is that a path through the net is created for each possible ordering of the transitions in the preset. This has the effect that each transition in the preset is given the chance to be the last one preventing the transitions in the postset from occurring. For illustration purposes, additional events c_0 and c_1 are added to the net to occur simultaneously with the two transitions associated with c. The timing behavior of c_0 and c_1 are shown graphically in Figure 7(b) and (c), respectively. The behavior of these two together is exactly the desired timing behavior of c. For ***n*** behavior places, the net is transformed to model the ***n!*** possible orderings of the ***n*** enabling events. While this transformation can lead to a substantial blowup in the net size, we have found that the value of ***n*** tends to be quite small in practical examples.

The transformation is more complicated in the case that one of the behavior places in the preset has multiple transitions in its postset. Consider a fragment of the orbital net from the SEL shown in Figure 8(a).

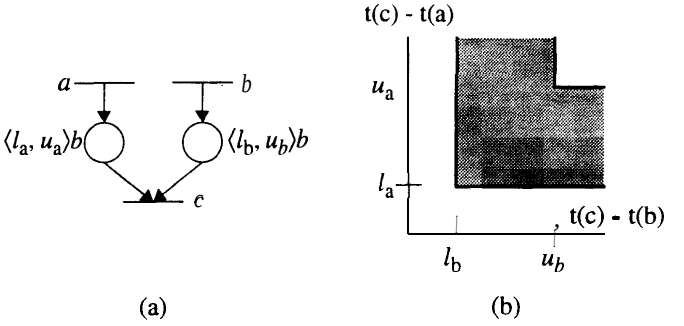


Figure 6: (a) Fragment of the orbital net that violates the single behavior place requirement; (b) graphical representation of the desired timing behavior.

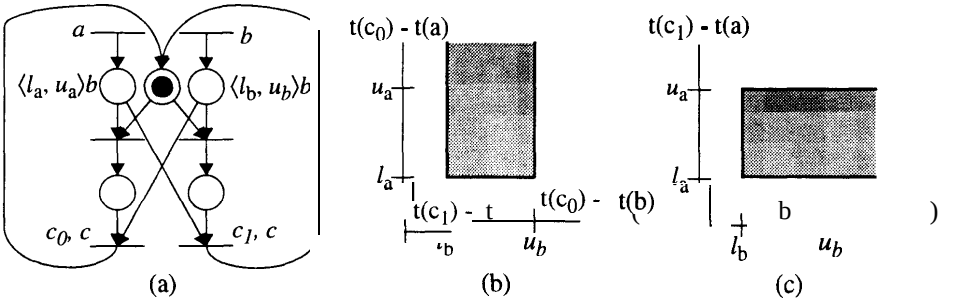


Figure 7: (a) Orbital net that satisfies the single behavior place requirement; graphical representation of the timing behavior of c_0 (b) and c_1 (c).

In this net, the behavior place in the postset of *data;* ↑ is shared by the transitions $out1_o$ ↑ and $out2_o$ ↑. In other words, if $out2_o$ ↑ occurs, the marking is removed before it can contribute to the firing of $out1_o$ ↑. In order to model this, the net is first transformed using the procedure described above for $out1_o$ ↑ and $out2_o$ ↑. Then, transitions are added to the part associated with *outl,* ↑ on $out2_o$ ↑ that reset the marking, and similarly transitions are added to the part associated with $out2_o$ ↑ on $out1_o$ ↑. A portion of the transformed net illustrating this is shown in Figure 8(b).

3 Timed state space exploration

The basic idea behind synthesis and verification methods that use explicit state space exploration is that, if the reachable state space is finite or has a finite representation, only a finite subset of the firing sequences needs to be considered to compute the complete set of reachable states. In orbital nets, the clocks associated with each marking can take on real values, so there are an infinite number of timed states. In order to perform explicit state space exploration, we must either group the timed states into a finite number of equivalence classes or sets, or restrict the set of values that the clocks can attain. In this section, we describe three previously proposed techniques for timed state space exploration: *unit-cubes, discrete-time,* and *geometric timing.* Then, we introduce our proposed technique, *partial order timing,* which improves upon the geometric methods by making use of concurrency and causality information.

3.1 Unit-cubes

Alur's unit-cube technique has the best known worst-case complexity for timed state space exploration of general timed systems [14]. This technique considers equivalence classes of timed states with the same integral clock values and a particular linear ordering of the fractional values of the clocks. For the case where

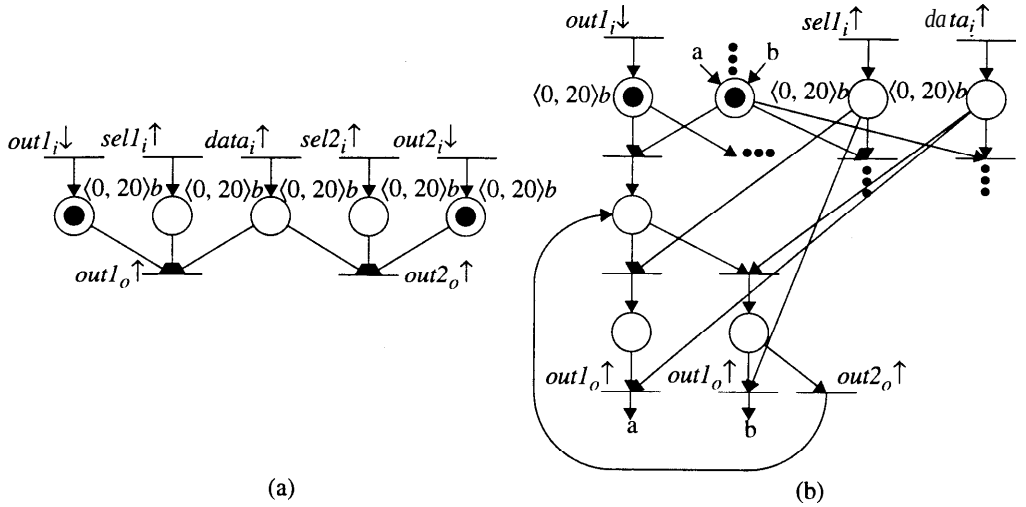


Figure 8: (a) Fragment of an orbital net with a behavior place that has multiple transitions in its postset; (b) part of the transformed orbital net which satisfies the single behavior place requirement (note "a" and "b" are shorthands for connections).

there are two marked places and two clocks clk_1 and clk_2, the equivalence classes are pictured in Figure 9(a); every point, line segment, and interior triangle is an equivalence class. Let us assume the number of distinct untimed states in an orbital net is $|S|$. If the maximum value of any timing requirement is k, and there are at most n marked places in the net in any state (this value is trivially bounded by the size of the safe net), the worst-case size of the state space for his method is asymptotically [9],

$$|S| \frac{n!}{\ln 2}\left(\frac{k}{\ln 2}\right)^n 4^{1/k}.$$

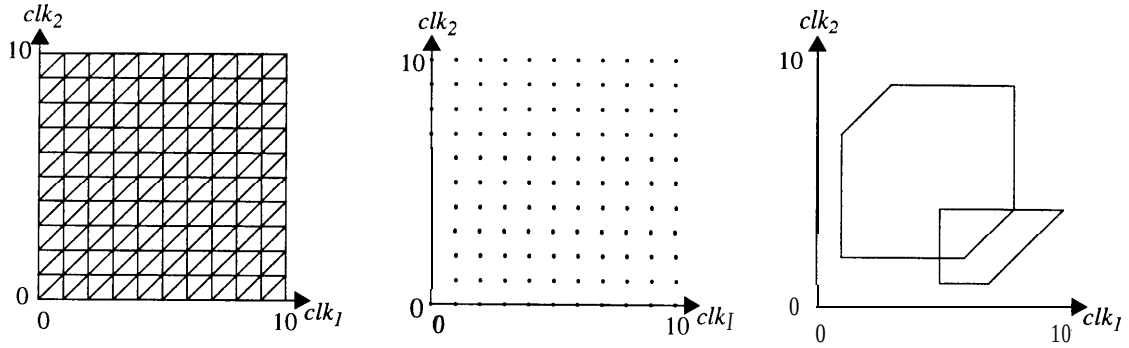


Figure 9: (a) Unit-cube, (b) discrete-time, and (c) geometric timing representations of the timed state space.

3.2 Discrete-time

It has been proven, however, that the general unit-cube technique is unnecessary for orbital nets since considering only integer event times gives a full characterization of the continuous-time behavior [9]. This proof is similar to one given by Henzinger, et. al. in [15] for timed transition systems. In other words, only every timed state associated with each discrete-time instance, represented as a point for the two-dimensional case in

Figure 9(b), needs to be considered. This was the technique employed by Burch for *verifying* timed circuits [8]. This technique has a worst-case state space size of $|S|(k+1)^n$ which is better than the unit-cube method by more than *n!*.

3.3 Geometric timing

Both unit-cubes and discrete-time, however, are of little more than theoretical interest, because the size of the state space increases exponentially with the concurrency in the net. In general, during analysis, every possible integer firing time must be considered for every transition in each state. For a circuit with timing values accurate to two significant digits, with up to six independent concurrent pending events, the state space is easily in excess of 10^{12} states-well beyond the capabilities of most finite-state synthesis and verification techniques.

In this section, we discuss *geometric timing*, a timing analysis technique that usually performs well in practice, even though the worst-case performance is much worse than either the unit-cube or the discrete-time approaches. Dill [16], Lewis [17], and Berthomieu and Diaz [18] originated geometric state space exploration, and it has become an active area of research [19, 20, 21].

3.3.1 Geometric regions

Rather than consider at each step a single discrete-time state, or a minimum equivalence class of timed states, the geometric timing method considers an infinite set of timed states in parallel. Specifically, convex geometric regions of timed states represented by upper and lower bounds on specific clock values and on the differences between pairs of specific clock values are used as the representation of the timed state space. The set of such constraints is usually represented by a matrix *A*, where the constraints on clocks $\{clk_1, \ldots, clk_n\}$ are of the form $clk_i - clk_j \leq a_{ji}$. A fictitious clock clk_0 that is always exactly zero is introduced so that upper and lower limits on a particular clock can be represented in the same form [16].

For any convex region that can be represented by such a matrix, there are many matrices that represent the same convex region. The process of *canonicalization* using Floyd's algorithm can be performed to yield a unique constraint matrix [16]. While in general Floyd's algorithm runs in time $O(n^3)$, since only incremental changes are made to the matrix during analysis, specializations of Floyd's algorithm that run in time $O(n^2)$ suffice [9]. Two sample regions are given in Figure 9(c).

3.3.2 State space exploration with geometric timing

Each geometric region can be considered as an infinite set of timed states which are operated on in parallel. In order to perform state space exploration using geometric timing, we redefine the operational semantics of orbital nets in terms of these geometric regions as opposed to individual timed states. We do not discuss the aspects of state space exploration that do not consider time, since they are the same in both cases. We describe how these operations work for a single step in a timed sequence, assuming it works for the predecessor sequence; the trivial base case and structural induction on sequences completes the proof that these operations work for all sequences.

In our original operational semantics, advancing time involves adding some number t to all clocks. For geometric regions, advancing time involves extruding the geometric region in the $clk_1 = clk_2 = \ldots = clk_n$ direction, subject to *max-advance*, which itself is a convex region.

Determining whether a particular transition is timed-enabled in our original operational semantics entails comparing the clocks with the timing requirements. With geometric regions, we determine the subset of the timed states in the region for which the particular transition is enabled. This can be performed by introducing the enabling conditions on the transition as additional constraints on the region and recanonicalizing. For

orbital nets, these conditions describe a convex region in the appropriate form, and it is easy to show that the intersection of two such convex regions is a convex region of the same form. Canonicalization by definition does not reduce the set of timed states represented.

After selecting an enabled transition, firing that transition involves removing some set of clocks and introducing new clocks initialized to zero. With geometric regions, removing these clocks involves projection of the system of constraints to eliminate a particular set of variables, and introducing new clocks is done by adding a new set of variables equal to zero.

3.3.3 Performance of geometric timing

While unit-cubes and discrete-time operate on timed firing sequences, geometric timing operates over untimed firing sequences. The function $untime(\alpha)$ returns the underlying *untimed firing sequence* from a given timed firing sequence by stripping the timing and removing any ϕ firings. For each untimed firing sequence α operated on by geometric timing, it calculates directly the full set of timed states reachable from all timed firing sequences β that satisfy $untime(\beta) = \alpha$. Thus, rather than separately considering every possible occurrence time for a particular transition in α during state space exploration, in one step the geometric region method considers all possible occurrence times.

State space exploration using geometric timing can be very efficient. However, some examples require an extremely large number of geometric regions such as the adverse example `adv4x4` 0 shown in Figure 10. While only having a single untimed state, standard geometric timing techniques generate an incredible 219,977,777 distinct geometric regions. This is more than either the number of discrete-time states or unit-cube equivalence classes.

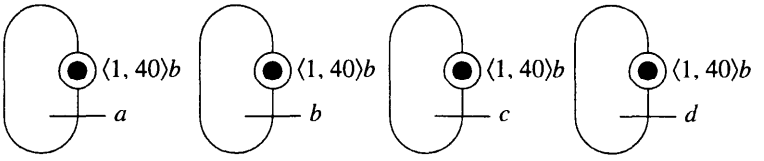


Figure 10: The adverse example `adv4x4` 0 with $n = 4$ and $k = 40$.

3.4 Partial order timing

In this section, we describe a new technique for timed state space exploration called *partial order timing.* This technique uses partial orders to improve the standard geometric timing approach for systems with concurrency. Recent work by Yoneda et. al. [22] also considered partial orders. Our work differs in that our formalism includes notions of specification, circuit composition, and receptiveness which enable us to perform efficient state space exploration on nontrivial timed circuit examples. To our knowledge, neither timed automata nor time Petri nets have been used in this fashion.

3.4.1 Concurrency, causality, and posets

The major source of blowup in the adverse example is the way the standard geometric timing algorithm calculates exactly the set of timed states reachable from a sequence of transition firings; the transition firings are linearly ordered, even if they are concurrent in the system being evaluated. That is, if two concurrent transitions start clocks, the constraints between the two clocks reflect the linear order that the transitions are fired in the original sequence. For example, when the geometric timing algorithm analyzes the untimed firing

sequence [a, b], it obtains the upper geometric region shown in Figure 11, and when the algorithm considers the sequence [*b*, a], it obtains the lower geometric region. In general, if there are n concurrent transitions that reset clocks visible in the resulting timed state, there are *n!* different sequences that need to be considered, each of which leads to a distinct geometric region. For this reason, it is important to distinguish the causal ordering of transitions from the non-causal ordering caused by the selection of a particular firing sequence.

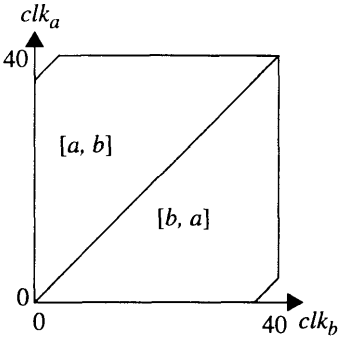


Figure 11: Geometric regions from the adverse example.

To solve this problem, we construct a partially order set, or *poset* for each untimed firing sequence which is represented with an acyclic, choice-free unfolding of the original orbital net. The poset reflects the causality and concurrency inherent in the firing sequence. Initially, the unfolded net representing the poset contains a single transition with places in its postset corresponding to each initially marked place. Transitions are added in the same order as they occur in the firing sequence. For each transition in the firing sequence, a correspondingly labeled transition is added to the unfolded net. A set of arcs into the transition are connected from the most recently added places in the unfolded net corresponding to places in the preset of the transition in the original orbital net. Finally, a new set of places corresponding to the places in the postset of the transition in the original net are added, and these places are connected to the new transition. Every place and every transition in the unfolded net, except the first, correspond to some place and some transition in the original net. Every place and every transition in the original net correspond to zero or more places and transitions in the unfolded net.

A poset explicitly represents the concurrency in a particular firing sequence. That is, a particular poset corresponds to many different firing sequences that differ only in the interleavings of concurrent transitions; every such firing sequence fires the same set of transitions and leads to the same final untimed state. For example, the poset represented with the unfolded net shown in Figure 12 corresponds both to the sequence [a, *b*] and to the sequence [b, a].

3.4.2 State space exploration with partial order timing

State space exploration proceeds just as it does for the previous methods based on sequences, except that, for each sequence, the algorithm constructs the corresponding unfolded net. With depth-first search, this is done incrementally. The algorithm also incrementally calculates a constraint matrix that stores the firing time relationship among the transitions. For each constraint place *p*, the constraint $t(\bullet p) \leq t(p\bullet)$ is introduced. For each behavior place *p* in the resulting unfolded net with a timing requirement of $\langle l, u\rangle b$, two constraints are introduced. The first reflects the minimum separation, $t(\bullet p) - t(p\bullet) \leq -l$. The second reflects the maximum separation, $t(p\bullet) - t(\bullet p) \leq$ u. All constraints introduced in this fashion for a given unfolded net must be satisfied. After canonicalizing this constraint matrix, it has produced a geometric region that represents the full set of reachable states for the poset corresponding to the unfolded net. Applying this procedure to the

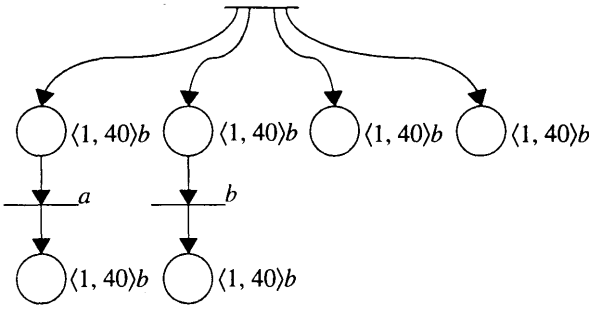


Figure 12: One poset from the adverse example.

unfolded net shown in Figure 12, we obtain at once the geometric region which encloses both regions shown in Figure 11.

While geometric timing operates on untimed firing sequences, partial order timing operates on posets. The function *poset* takes an untimed firing sequence and returns the corresponding unfolded net. For each untimed firing sequence α operated on by the partial order technique, it calculates directly the full set of timed states reachable from any timed firing sequence β such that $poset(untime(\beta)) = poset(\alpha)$. Thus, rather than separately considering every interleaving of concurrent transitions, in one step the partial order method considers all possible interleavings. For untimed state space exploration, different interleavings result in the same state. For timed state space exploration, different interleavings usually result in different sets of timed states, with different future behavior, leading to a combinatorial explosion of timed regions for each untimed state. Representing, as a single constraint matrix, the union of all timed states reachable from all possible interleavings, therefore, dramatically reduces the size of the state space representation. In fact, the partial order method typically reduces the average number of timed regions for each untimed state to a value close to one. For example, partial order timing applied to the adverse example in Figure 10 obtains exactly one geometric region corresponding to the one untimed state.

3.4.3 Efficiency considerations

The number of transitions in the unfolded net is equal to the length of the firing sequence plus one, and it increases with the depth of our search. Calculating the minimum separations between the occurrence times in the unfolded net, even with our incremental $O(n^2)$ approach, becomes prohibitively expensive as the firing sequence lengthens. In addition, the algorithm needs a constraint matrix for each step; this would require a tremendous amount of storage during depth-first search.

To keep n bounded as the depth of our search increases, the algorithm determines what prefix, if any, of the unfolded net can safely be ignored. The algorithm can eliminate any transitions that no longer affect future calculations. In general, the algorithm can eliminate a variable from any set of equations or inequalities whenever it has produced the full set of equations or inequalities that use that variable. Since all constraints introduced through the firing of a transition are associated with places connecting the new transition to the old, once a transition in the unfolded net no longer has any places in its postset which do not have a transition in their postset, it is eliminated from our constraint matrix. Thus, our n is—at most-the number of marked places in the original net at any given time, plus one for the current transition.

Because the number of geometric regions is typically small, a further optimization is possible. Rather than backtracking only when an identical geometric region is found, our search can backtrack whenever a new geometric region is a subset of a previously seen geometric region. Comparing two geometric regions

for inclusion can be performed in 0(n^2) time.

4 Synthesis procedure

Synthesis is the process of transforming a specification into a circuit implementation. Our synthesis procedure *begins* with a THSE specification from which a hazard-free timed circuit implementation is generated using only basic gates such as AND gates, OR gates, and C-elements. After the specification is compiled to an orbital net representation as described above, the partial order timing algorithm is used to find the set of reachable states. From the resulting state graph, there are several different approaches that could be used to obtain a gate-level timed circuit implementation. The first approach is to use a traditional boolean minimization technique directly. We demonstrate, however, that when mapping the resulting implementation to basic gates, it may result in a hazardous implementation. Another approach is to split the design of the rising and falling transitions to obtain a *generalized C-element* implementation [1] and decompose it to basic gates. This technique alleviates some of the hazard problems, but we demonstrate that it may still be hazardous when mapped, to basic gates. We take a *standard C-implementation* approach in which each rising and falling region for each output signal is implemented using a single AND gate, or *cube,* which must satisfy certain constraints. A covering problem is setup and solved to find an optimal implementation for each region. When all the regions are merged, the resulting implementation is guaranteed to be a hazard-free gate-level timed circuit.

4.1 Finding the state graph

Our timing analysis algorithm operates on the orbital net representation to find the reachable state space represented as a *state graph* (SG). A SG is a graph in which its vertices are untimed states and its edges are possible *state transitions.*

A state graph is modeled by the tuple (I, 0, Φ, $\Gamma\rangle$ where I is the set of input signals, 0 is the set of output signals, Φ is the set of states, and $\Gamma \subseteq \Phi$ x Φ is the set of edges. For each untimed state s, there is a corresponding labeling function s : $I \cup 0 \rightarrow \{0, R, 1, F\}$ which returns the value of each signal and whether it is untimed-enabled, i.e.,

$$s(u) \equiv \begin{cases} 0 & \text{if u is stable low in s} \\ R & \text{if u is untimed-enabled to rise in s} \\ 1 & \text{if u is stable high in } s \\ F & \text{if u is untimed-enabled to fall in s.} \end{cases}$$

It is useful to also define a function *val* which strips the excitation information, i.e.,

$$val(x) \equiv \begin{cases} 0 & \text{if } x = 0 \text{ or } x = R \\ 1 & \text{if } x = 1 \text{ or } x = F. \end{cases}$$

Traditional definitions of state labeling functions have not included the enabling of signals as it can usually be inferred from the set of state transitions. In timed circuits, however, it is possible that a signal is untimed-enabled but not timed-enabled in a given state. In this case, there would be no state transition out of that state in which that signal fired, and thus, it would not be possible to infer from the state graph that the signal is untimed-enabled.

A state graph is defined to be well-formed if it is strongly connected and for any state transition (s, s ′) in Γ, the value of exactly one enabled signal in s changes to a new value in s'. A state transition (s, s') and the signal v that differs in value is denoted as follows: $s \xrightarrow{v}$ s'. Our synthesis procedure also requires that the state graph be *complete state coded,* defined to be that if for any two states in which all signals have the same value, any output signal untimed-enabled in one state is also untimed-enabled in the other. It has been

reported that adding state variables can transform an arbitrary state graph into one that satisfies complete state coding [23, 24, 25]. These approaches, however, may be conservative when timing is considered. Therefore, we believe adding state variables in a timed specification is an interesting open research problem.

While obtaining a state graph, there may be violations of some constraint timing requirements. For our nets, this typically occurs if an ordering assumed at the specification level is not guaranteed by the specified behavior. If there are any constraint violations, the specification is automatically modified, if possible, to solve the problem. If all transitions in the postset of the violating constraint place are output signals, our procedure transforms it to a behavior place with a default timing requirement. If any transition in the postset of the violating constraint place is an input, it must instead slow down an earlier transition by adding a behavior place between the transitions in the preset and the first output transition that precedes the transitions in the postset.

For the SEL, there are four constraint violations on the places between $xfer_i\uparrow$ and $data_i\uparrow$, $xfer_i\uparrow$ and $sel_o\uparrow$, $data_i\uparrow$ and $out1_o\uparrow$, and $data_i\uparrow$ and $out2_o\uparrow$. All of these violations have output signals in the postset, so they are simply changed to behavior places with a default timing requirement of $\langle 0, 20\rangle b$. The final state graph obtained for the SEL contains 53 states. A state graph generated ignoring all the timing information contains 256 states. As shown later, this larger state graph leads to a larger circuit implementation.

4.2 Boolean minimization technique

After obtaining a state graph, we could apply a traditional Boolean minimization technique to find an implementation. Using this technique, the state space is partitioned into an *on-set,* an *off-set,* and a *don 't-care-set.* Then, a Boolean minimization program, such as espresso [26] can be used to find the optimal sum-of-products representation. For our designs, a minimization problem would be setup for each output signal u with the on-set containing each state s in which the signal is enabled to rise or is stable high (i.e., s(u) = R or s(u) = 1), the off-set containing each state s in which the signal is enabled to fall or is stable low (i.e., s(u) = F or s(u) = 0), and the don't-care-set containing all unreachable states (i.e., $\beta^{I\cup O} - \Phi$).

Applying this technique to the signal $out2_o$ from the SEL results in the Boolean equation:

$$out2_o = (data_i \wedge sel2_i \wedge \neg out2_i) \vee (data_i \wedge out2_o) \vee (\neg xfer_o \wedge out2_o)$$

In order to guarantee correctness, Chu [2] and Meng [3] assumed that the logic equation for each output signal could be implemented directly with a single complex *atomic* gate. In other words, each signal is built with an instantaneous function block with a delay element connected to its output. Unfortunately, if the equation is mapped to basic gates and the delays of these gates are considered individually, the implementation may be hazardous. For example, the equation for $out2_o$ could be implemented directly as a sum-of-products as shown in Figure 13. If the 3-input AND and OR gates (gates 1 and 4) are assumed to have a delay of $\langle 2, 5\rangle$ while the 2-input AND gates (gates 2 and 3) have a delay of $\langle 2, 3\rangle$, this implementation is hazard-free. However, if the upper bound of the delay on the 2-input AND gates increases to 4 or more time units, this circuit is now hazardous. The segment of the state graph to the left illustrates a sequence of transitions which cause a hazard. Essentially, after gate 1 has caused $out2_o$ to rise, it has the potential of being shut off again before gates 2 or 3 can come on to hold the state.

In order to solve this problem, Lavagno [6] first mapped the logic equations ignoring hazards using standard synchronous techniques, then added delay elements where necessary to remove any potential hazards. This technique, however, not only adds additional overhead in terms of area and delay, but the resulting circuits may not be very reliable due to the difficultly in designing delay elements with accurate timing.

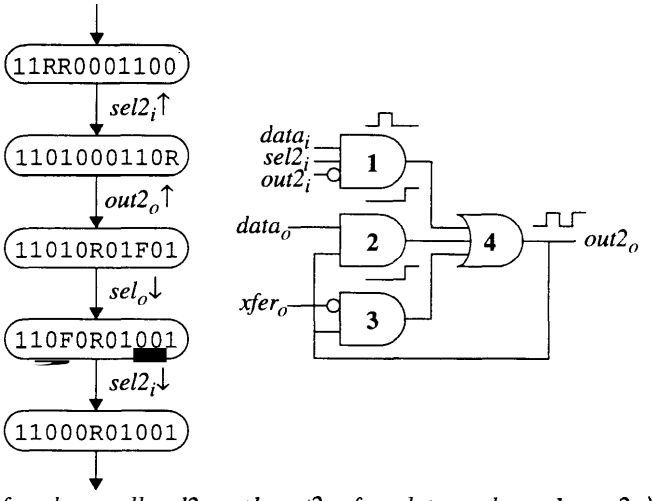


Figure 13: A hazardous sum-of-products implementation of $out2_o$ from the SEL.

4.3 Generalized C-element technique

Another implementation strategy originally proposed by Martin [1] is to use generalized C-elements as the basic building blocks. This is also the technique used in our earlier work [7]. In this technique, the implementation of the set and reset of a signal are decoupled. The basic structure is depicted in Figure 14(a) in which the upper sum-of-products represents the logic for the set, the lower sum-of-products represents the logic for the reset, and the result is merged with a C-element. This can be implemented directly in CMOS as a single compact gate with weak-feedback as shown in Figure 14(b) or as a fully-static gate as shown in Figure 14(c).

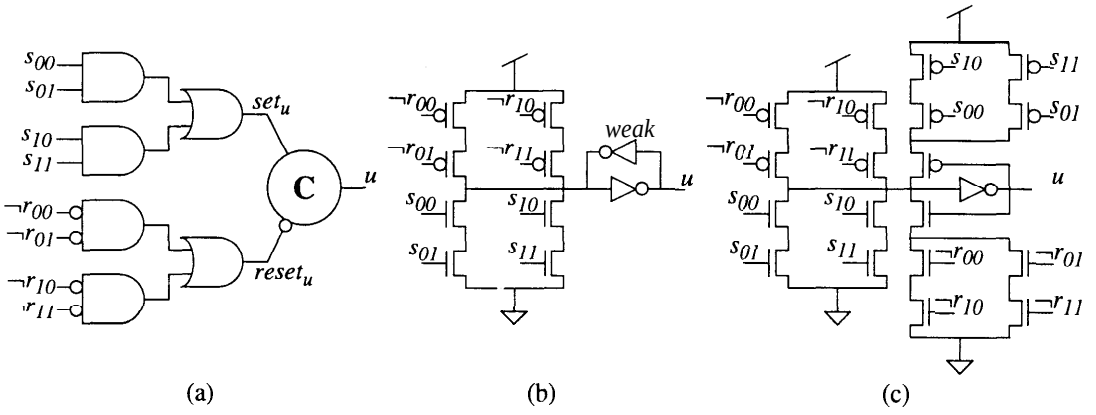


Figure 14: (a) The generalized C-element configuration and (b) weak-feedback and (c) fully-static CMOS implementations.

Using a procedure similar to the one described in [7], we obtain a generalized C-element implementation for the signal $out2_o$ shown in Figure 15. While this could be implemented with a single generalized C-element, a gate-level implementation would be composed of a 3-input AND gate, a 2-input AND gate, and a C-element. Although this implementation no longer has the hazard associated with the SG fragment in Figure 13, it now has a hazard illustrated with the state graph shown to the left in which the reset AND gate glitches while the output is stable low. For the specified delays, it can be shown that the hazard does not propagate to the output, but given appropriate delays this hazard may propagate [11]. To address this problem, after a generalized

C-element implementation is produced and decomposed to basic gates, the design could be back-annotated with delays from the gate library, and the circuit could be verified. While this may often work, it is not clear what to do in the cases in which a hazard does exist. Also, a hazard is a spurious transition which wastes power and does no useful work. In a power efficient implementation, it is desirable to have logic which is hazard-free both internally and externally.

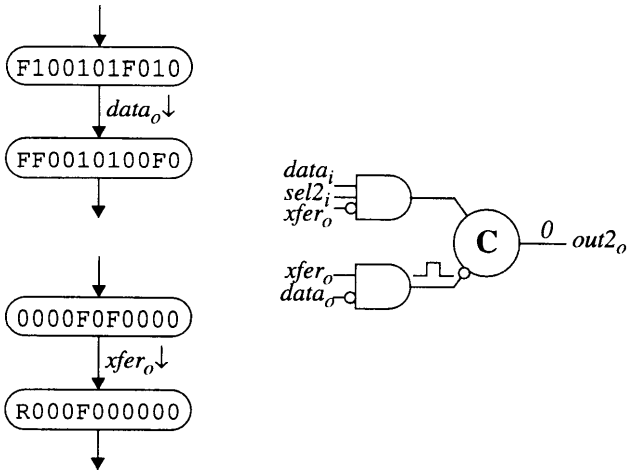


Figure 15: A hazardous gate-level implementation of $out2_o$ from the SEL.

4.4 Standard C-implementation technique

To avoid the hazard concerns discussed above, our approach obtains a gate-level implementation by first decomposing the design into a set of rising and falling regions which are each implemented using a single cube. While the general structure of the standard C-implementation is similar to the generalized C-element structure shown in Figure 14, each cube in the set or reset block must satisfy certain constraints to guarantee that the merged implementation is a gate-level hazard-free circuit. The approach is conservative in that timing analysis may show that the decomposed generalized C-element implementation is sufficient, but the overhead required tends to be small to get a safe implementation that is free of internal hazards.

4.4.1 Excitation regions and quiescent states

In order to obtain our gate-level implementation, the SG is decomposed for each output signal into a collection of *excitation regions.* An excitation region for the output signal u is a maximally connected set of states in which the signal is enabled to change to a given value (i.e., s(u) = *R* or s(u) = *F).* If the signal is rising in the region (i.e., s(u) = *R),* it is called a *set region,* otherwise the region is called a *reset region. The* excitation regions for a signal u are indexed with the variable k and the k^{th} excitation region of signal u is denoted $ER(u, k)$. Typically, different excitation regions correspond to different output signal transitions in a higher-level specification. For example, there are two set regions for the signal $xfer_o$ in the SEL which correspond to the two instances of $xfer_o\uparrow$ in the THSE in Figure 2.

For each excitation region, there is an associated set of *quiescent,* or stable, states $QS(u, k)$. For a set region $ER(u, k)$, it is the states where the signal is stable high (i.e., $QS(u, k)$ = {s $\in \Phi$ | s(u) = 1}), and for a reset region, it is the states where the signal is stable low (i.e., $QS(u, k)$ = {s $\in \Phi$ | s(u) = O}).

4.4.2 Correct covers

In our gate-level implementation, each excitation region is implemented with a single cube corresponding to a *correct cover* of the excitation region. The cover of an excitation region $C(u,k)$ is a set of states for which the corresponding cube in the implementation evaluates to one. A cover is a correct cover if it satisfies two conditions. First, it must satisfy the *covering constraint* which says that the reachable states in the cover must include the entire excitation region but must not include any states outside of the union of the excitation region and associated quiescent states, i.e.,

$$ER(u,k) \subseteq [C(u,k) \cap \Phi] \subseteq [ER(u,k) \cup QS(u,k)].$$

Second, it must satisfy the *entrance constraint* which says that a correct cover must only be entered through excitation region states, i.e.,

$$[(s,s') \in \Gamma \text{ A } s \notin C(u,k) \text{ A } s' \in C(u,k)] \Rightarrow s' \in ER(u,k).$$

The definition of correct covers is based on the definition given for speed-independent circuits in [27]. This definition differs slightly from the one in [27] in that $QS(u,k)$ does not need to be a maximally connected set of states, but it is easy to show this condition is made redundant by the entrance constraint. Extending the proof for the speed-independent case, it can be shown that these conditions ensure when the specified timing assumptions are met that the resulting gate-level timed circuit implementation is hazard-free.

4.4.3 Finding enabled cubes and trigger cubes

Since each region is implemented with a single cube, to obtain a hazard-free implementation, all literals in the cube must correspond to signals that are *stable,* i.e., constant throughout the excitation region. Otherwise, the single-cube cover would not cover all excitation region states. When a single-cube cover exists, an excitation region $ER(u,k)$ can be sufficiently approximated using a cube called an *enabled cube,* denoted $EC(u,k)$, defined on each signal v as follows:

$$EC(u,k)(v) \equiv \begin{cases} 0 & \text{if } \forall s \in ER(u,k)\,[val(s(v)) = 0] \\ 1 & \text{if } \forall s \in ER(u,k)\,[val(s(v)) = 1] \\ X & \text{otherwise} \end{cases}$$

If a signal has a value of 0 or 1 in the enabled cube, it can be used in the cube implementing the region. A cube, such as the enabled cube, implicitly represents a set of states in the obvious way. The set of states represented by the enabled cube is always a superset of the set of excitation region states (i.e., $EC(u,k) \supseteq ER(u,k)$).

Each cube in the implementation is composed of *trigger signals* and *context signals.* For a given excitation region, a trigger signal is a signal whose firing can cause the circuit to enter the excitation region while any non-trigger signal which is stable in the excitation region can potentially be a context signal. The set of trigger signals for an excitation region $ER(u,k)$ can also be represented with a cube called a *trigger cube* $TC(u, v)$ defined as follows for each signal v:

$$TC(u,\mathbf{k})(\mathbf{w}) \equiv \begin{cases} val(s'(v)) & \text{If } \exists s,s'\,[(s \xrightarrow{v} s') \text{ A } (s \notin ER(u,k)) \text{ A } (s' \in ER(u,k))] \\ X & \text{otherwise} \end{cases}$$

In order for our synthesis procedure to generate a circuit, the cover of each excitation region must contain all its trigger signals (i.e., $C(u,k) \subseteq TC(u,k)$). Since only stable signals can be included, a necessary condition for our algorithm to produce an implementation is that all trigger signals be stable (i.e., $EC(u,k) \subseteq TC(u,k)$). If a trigger signal is not stable then we must either constrain concurrency [3], add state variables [28], or use a more general algorithm [27].

The enabled cubes and trigger cubes are easily found with a single pass through the state graph. Table 1 shows the enabled cubes and trigger cubes corresponding to all the excitation regions in the SEL.

Table 1: Enabled cubes and trigger cubes for the SEL.

u, k	*set/reset*	$EC(u,k)$	$TC(u,k)$
$xfer_o, 0$	*set*	11X0100X010	XXXX1XXXXXX
$xfer_o, 1$	*set*	110X010X001	XXXXX1XXXXX
$xfer_o, 2$	*reset*	0X00XX10000	*oxxxxxxxxxx*
$data_o, 0$	*set*	*10000000x00*	*lxxxxxxxxxx*
$data_o, 1$	*reset*	11X010X1010	XXXX1XXXXXX
$data_o, 2$	*reset*	110X01X1001	XXXXX1XXXXX
$sel_o, 0$	*set*	*1000000x000*	*lxxxxxxxxxx*
$sel_o, 1$	*reset*	11100001110	XXXXXXXXX1X
$sel_o, 2$	*reset*	11010001101	XXXXXXXXXX1
$out1_o, 0$	*set*	11100001100	X11XXXXXXXX
$out1_o, 1$	*reset*	1XX01010010	XXXXXX10XXX
$out2_o, 0$	*set*	~11010001100	X1X1XXXXXXX
$out2_o, 1$	*reset*	1X0X0110001	XXXXXX10XXX

$\langle xfer_i, data_i, sell_i, sel2_i, out1_i, out2_i, xfer_o, data_o, sel_o, out1_o, out2_o \rangle$

4.4.4 Finding an optimal correct cover

Our procedure to find a correct cover begins with a cube consisting only of the trigger signals (i.e., C(u, *k) = TC*(u, k)*).* If this cover contains no *conflicts*, i.e., states that violate either the covering or entrance constraint, we are done. This, however, is often not the case, and context signals must be added to the cube to remove any conflicting states. For each conflict detected, the procedure determines the choices of context signals which would exclude the conflicting state. Finding the smallest set of context signals to resolve all conflicts is a covering problem. Due to the implication in the entrance constraint, inclusion of certain context signals may introduce additional conflicts which must be resolved. Therefore, the covering problem is *binate*.

To solve our binate covering problem, we create a *covering and closure (CC) table* [29] for each region. While other techniques exist to find binate covers such as those described in [30, 31], the CC table is simple and facilitates presentation. There is a row in the CC table for each context signal, and there is a column for each conflict and each conflict that could potentially arise from a context rule choice. An entry in the table contains a cross (x) if the context signal resolves the conflict. An entry in the table contains a dot (o) if the inclusion of the context signal would require the conflict to be resolved.

To construct the table for a given excitation region $ER(u, k)$, the procedure first finds all states in the initial cover (i.e., $TC(u, k)$) which conflict with the covering constraint. In other words, a conflict exists in a state s in $TC(u, k)$ if the signal u has the same value but is not enabled (i.e., s(u) = 0 for a set region or s(u) = 1 for a reset region), is enabled in the opposite direction (i.e., s(u) = *F* for a set region or s(u) = *R* for a reset region), or is enabled in the same direction but the state is not in the current excitation region (i.e., s(u) = *R* for a set region or s(u) = *F* for a reset region and $s \notin EC(u, k)$). If a conflict exists, the procedure adds a new column to the table with a cross in each row corresponding to a context signal v that would exclude the conflicting state (i.e., $EC(u, k)(w) = \neg val(s(v))$).

The next step in the table construction is to find all state transitions which conflict with the entrance constraint in the initial cover or may conflict due to a context signal choice. For any state transition $s \xrightarrow{v} s'$, this is possible when s is not in the excitation region (i.e., $s \notin EC(u, k)$), s' is a quiescent state (i.e., s'(v) = 1 for a set region and s'(v) = 0 for a reset region), s' is in the initial cover (i.e., $s' \in TC(u, k)$), and *v* excludes s (i.e., $EC(u, k)(w) = \neg val(s(v))$). For each entrance conflict or potential entrance conflict detected, the

procedure adds a new column to the table again with a cross in each row corresponding to a context signal that would exclude the conflicting state. If the signal v in the state transition is a context signal, the state s' only needs to be excluded if v is included in the cover. This implication is represented with a dot being placed in the row corresponding to the signal v.

If a conflict is detected for which there is no context signal to resolve it, the CC table construction fails. In this case, as with non-stable trigger signals, it is necessary to constrain concurrency, add state variables, or use a more general algorithm.

In a single pass through the state graph, all the CC tables can be constructed. When implementing ($out2_o$, 1) from the SEL, no covering conflicts are detected. This is not surprising since our complex-gate implementation of this region only contained the trigger signals $xfer_o$ and $\neg data_o$. There are, however, entrance conflicts which are shown in the CC table in Table 2.

Table 2: The CC table for the region ($out2_o$, 1) from the SEL.

Signal	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$xfer_i$		×		×	×								×		
$sel1_i$	×	o	×		o	×	×	o		o		o	×		o
$out1_i$	×	×	o	o		×	×	×	×	×	o		o	×	×
$out2_i$	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×
$xfer_o$															
$data_o$															
sel_o															
$out1_o$	o					×	o	×	o					×	×
$out2_o$	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×

The last step is to find the smallest set of context signals to implement each excitation region by solving the CC tables that are constructed. The CC tables are solved using standard reduction rules [29] given below:

Rule 1: (Select essential rows) If a column contains only a single cross and blanks elsewhere, then the row with the cross must be selected. The row is deleted together with all columns in which it has crosses.

Rule 2: (Remove columns with only dots) If a column has only a single dot and blanks elsewhere, the row containing the dot must be deleted together with all columns in which it has dots.

Rule 3: (Remove dominating columns) A column C_j dominates a column C_i if it has all the crosses and dots of C_i. If C_j dominates C_i, then C_j is deleted.

Rule 4: (Remove dominated rows) A row R_i dominates a row R_j if it (a) has all the crosses of R_j; and (b) for every column C_p in which R_i has a dot, either R_j has a dot in C_p or there exists a column C_q in which R_j has a dot, such that, disregarding the entries in rows R_i and R_j, C_p dominates C,. If R_i dominates R_j, then R_j is deleted together with all columns in which it has dots.

Rule 5: (Remove rows with only dots) If a row only has dots, then the row is deleted together with all columns in which it has dots.

It is important to note that when applying rule 4, two rows may mutually dominate each other. These ties are resolved by picking the rule that provides symmetry between different regions of the same signal. This symmetry often leads to gates being shared between regions. The table is completely solved when all columns are eliminated, and the context signals are those corresponding to the essential rows selected by Rule 1. While in practice these reduction rules are often sufficient to solve the table, some tables may be *cyclic.* To solve the cyclic table, we use a branch and bound method.

For the SEL, the CC table for ($out2_o$, 1) is reduced to the one shown in the left-half of Table 3 after

removing dominating columns. The CC table is further reduced to the one shown in the right-half of Table 3 after removing dominated rows. This leaves us with a choice of using either $out2_i$ or $out2_o$ as a context signal. In this case, they are equivalent, and we arbitrarily select $out2_i$.

Table 3: The CC table for the region ($out2_o$, 1) from the SEL after removing dominating columns.

Signal	9	11	12	14	14
$xfer_i$					
$sel1_i$			o		
$out1_i$	×	o		×	
$out2_i$	×	×	×	×	×
$xfer_o$					
$data_o$					
sel_o					
$out1_o$	o			×	
$out2_o$	x	x	x	x	x

For the SEL, we derive a gate-level timed circuit implementation with 27 literals shown in Figure 16(a). If all the timing information is ignored, we obtain a gate-level speed-independent circuit implementation with 44 literals shown in Figure 16(b). Besides being nearly 40 percent smaller, the timed circuit has reduced latency since it requires gates with at most 3-inputs while the speed-independent circuit requires many large gates including one with 6-inputs.

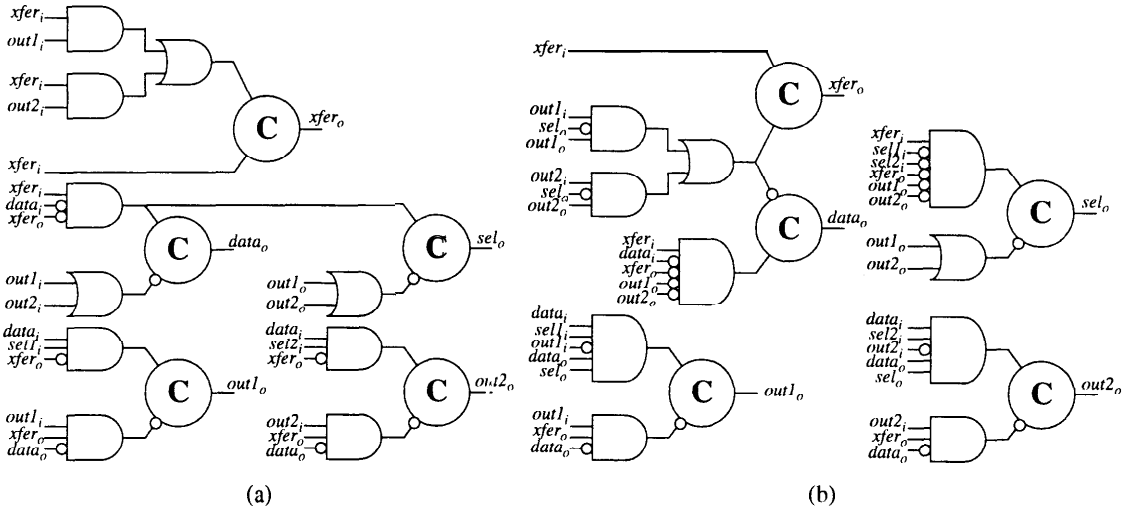


Figure 16: Gate-level (a) timed and (b) speed-independent circuits for the SEL.

4.5 Experimental results

The synthesis procedure described in this paper has been fully automated within the CAD tool ATACS. We have applied this procedure to several examples and compared our results with designs produced using other asynchronous design methodologies including Beerel's speed-independent method (SYN) [27], Lavagno's method which adds delay elements to remove hazards (SIS) [6], and Yun's burst-mode method (3D) [32]. Our synthesis procedure results in timed designs which are not only more efficient than designs produced by

these other asynchronous design methodologies but also more efficient than comparable synchronous designs. This section describes two examples in detail, an MMU controller and a DRAM controller, and summarizes the rest of our results.

4.5.1 MMU controller

In [7], we presented a complex-gate implementation of one of the six possible cycles for an asynchronous memory management unit (MMU) [33]. Previously, our synthesis procedure could not synthesize the complete design as it contains events concurrent with the environment making a choice. The specification for the MMU controller process is shown in Figure 17. From this specification, our synthesis procedure obtains a state graph which contains 187 states. From the state graph, the procedure obtains a gate-level timed circuit implementation with *62* literals depicted in Figure 18(a) using only basic gates with at most 3-inputs. For a gate-level speed-independent circuit implementation, the state graph explodes to 23,296 states resulting in the circuit implementation shown in Figure 18(b) that is not only significantly larger, 114 literals, but also significantly slower since it requires gates as large as 8 inputs!

module MMU;

$$
\begin{array}{l}
*[[\ [MDl_i\uparrow] \rightarrow (([RA_i\downarrow];RA_o\uparrow)\ ||\ ([B1_i\downarrow \vee B2_i\downarrow \vee B3_i\downarrow];B_o\uparrow));[RA_i\uparrow]; \\
\quad [\ [B1_i\uparrow \wedge LSR_i\downarrow];\rightarrow LSR_o\uparrow;(RA_o\downarrow\ ||\ B_o\downarrow);[LSR_i\uparrow];MDl_o\uparrow;LSR_o\downarrow;[MDl_i\downarrow];MDl_o\downarrow \\
\quad |\ [B2_i\uparrow \wedge LSW_i\downarrow];\rightarrow LSW_o\uparrow;(RA_o\downarrow;||\ B_o\downarrow);[LSW_i\uparrow];MDl_o\uparrow;LSW_o\downarrow;[MDl_i\downarrow];MDl_o\downarrow \\
\quad |\ [B3_i\uparrow \wedge MSl_i\downarrow];\rightarrow MSl_o\uparrow;(RA_o\downarrow;||\ B_o\downarrow);[MSl_i\uparrow];MDl_o\uparrow;MSl_o\downarrow;[MDl_i\downarrow];MDl_o\downarrow \\
\quad] \\
|\ [MDs_i\uparrow] \rightarrow (([WA;\downarrow];WA,\uparrow)\ ||\ ([B1_i\downarrow \vee B2_i\downarrow \vee B3_i\downarrow];B_o\uparrow));[WA_i\uparrow]; \\
\quad [\ [B1_i\uparrow \wedge SSR_i\downarrow];\rightarrow SSR_o\uparrow;(WA_o\downarrow||B_o\downarrow);[SSR_i\uparrow];MDs_o\uparrow;SSR_o\downarrow;[MDs_i\downarrow];MDs_o\downarrow \\
\quad |\ [B2_i\uparrow \wedge SSW_i\downarrow];\rightarrow SSW_o\uparrow;(WA,\downarrow;||B_o\downarrow);[SSW_i\uparrow];MDs_o\uparrow;SSW_o\downarrow;[MDs_i\downarrow];MDs_o\downarrow \\
\quad |\ [B3_i\uparrow \wedge MSs_i\downarrow];\rightarrow MSs_o\uparrow;(WA_o\downarrow;||B_o\downarrow);[MSs_i\uparrow];MDs_o\uparrow;MSs_o\downarrow;[MDs_i\downarrow];MDs_o\downarrow \\
\quad] \\
]| \\
||\ \text{etc.}
\end{array}
$$

Figure 17: Part of the THSE for an MMU.

4.5.2 DRAM controller

Our second example is a DRAM controller which is an interface between a microprocessor and a DRAM array. Typically, a DRAM controller is implemented as a synchronous circuit. Since a DRAM controller must interface with a synchronous environment, it cannot be implemented as a speed-independent asynchronous circuit, but it can be implemented as a timed circuit that satisfies certain timing constraints.

'A block diagram for the entire DRAM controller is shown in Figure 19. Our specification shown in Figure 20 is derived from a burst-mode specification given in [34]. The DRAM controller has three possible modes of operation: *read, write,* and *refresh.* In [7], due to this choice in the specification, in order to derive a timed circuit implementation, the specification had to be unwound to make it deterministic. Also, the timed circuit derived required complex-gates, and thus, it is not hazard-free at the gate-level. The synthesis procedure reported in this paper can be directly applied to the original specification with choice to produce the gate-level hazard-free timed circuit shown in Figure 21(a). While the two implementations have a different structure, they are equivalent in terms of literal count (38 literals each) before optimizations, so there is no cost in achieving hazard-freedom at the gate-level. A synchronous implementation of the DRAM controller shown in Figure 21(b) is generated using Berkeley's synchronous synthesis program SIS [35]. Surprisingly, our timed

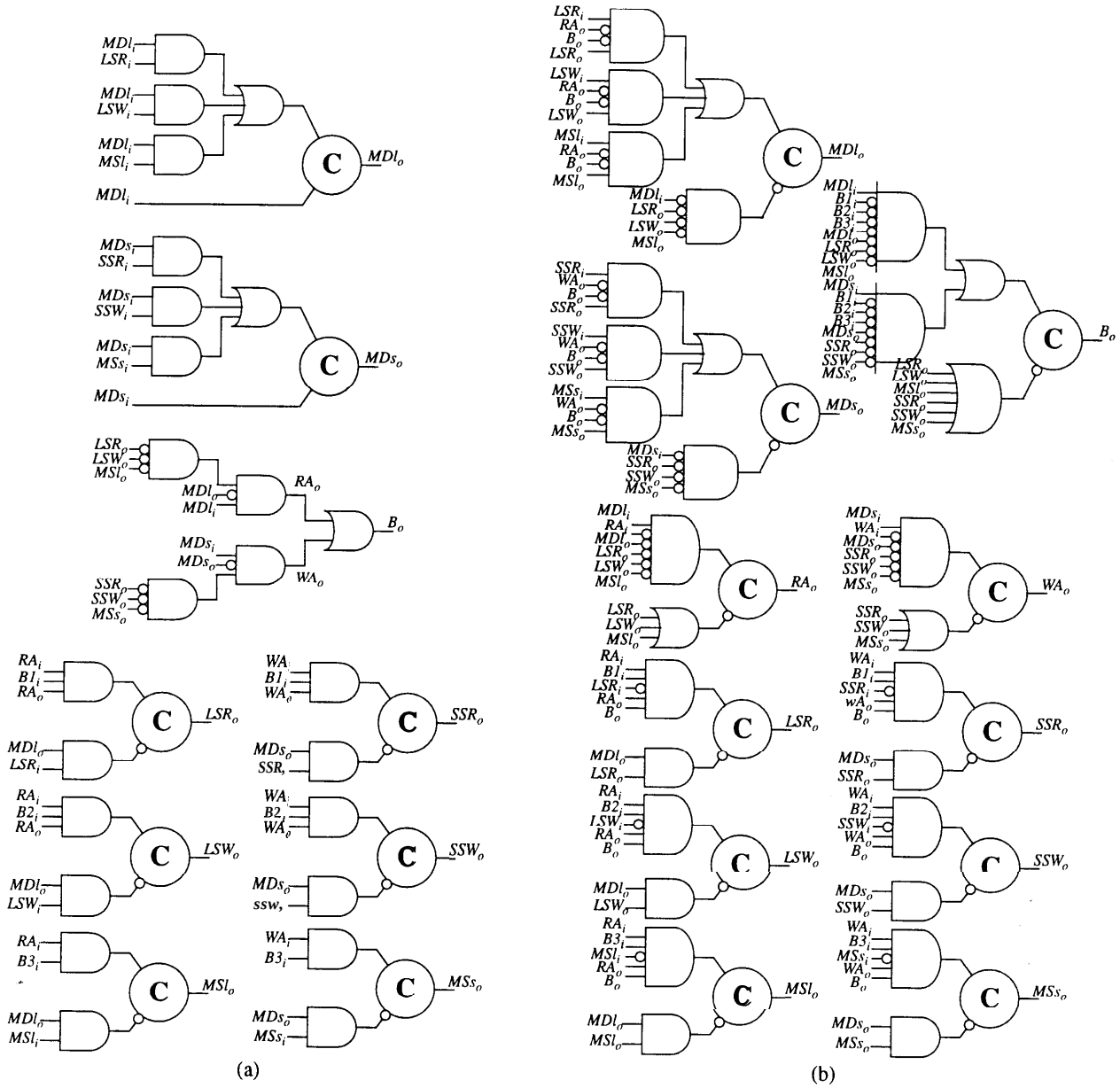


Figure 18: Gate-level (a) timed and (b) speed-independent circuits for the MMU.

design is about 40 percent smaller and 30 percent faster. This result comes from the sequential don't-care information that is taken into account by the asynchronous nature of our synthesis procedure. There is also a significant improvement in power consumption since our timed design produces no spurious transitions.

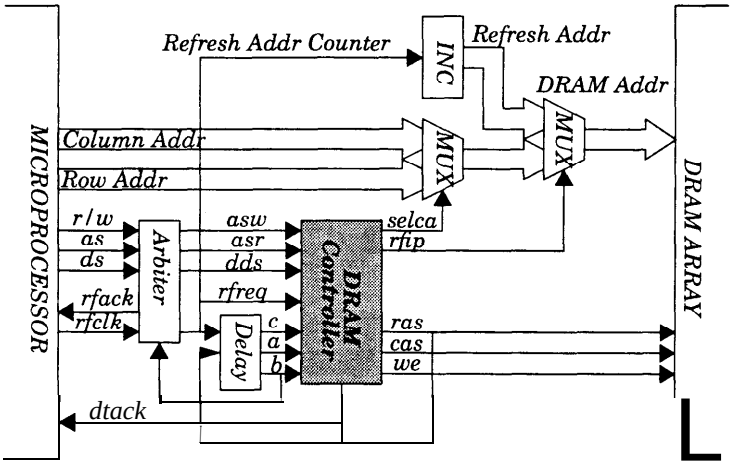


Figure 19: Block diagram for a DRAM interface.

module DRAM;

$$
\begin{aligned}
*\,[[\,[rfreq\downarrow] \rightarrow rfip\uparrow; [c\downarrow]; ras\downarrow; [a\downarrow \wedge b\downarrow \wedge rfreq\uparrow]; (rfip\downarrow \;||\; ras\uparrow); [a\uparrow \wedge b\uparrow \wedge c\uparrow] \\
\;\;|\;[asw\downarrow] \rightarrow ras\downarrow; [a\downarrow]; (dtack\downarrow || we\downarrow || selca\uparrow); [b\downarrow \mathrm{A}\; dds\downarrow]; cas\downarrow; \\
\qquad\qquad [asw\uparrow \mathrm{A}\; dds\uparrow]; (ras\uparrow || cas\uparrow || dtack\uparrow || we\uparrow || selca\downarrow); [a\uparrow \mathrm{A}\; b\uparrow] \\
\;\;|\;[asr\downarrow \mathrm{A}\; dds\downarrow] \rightarrow ras\downarrow; [a\downarrow]; (dtack\downarrow || selca\uparrow); [b\downarrow]; cas\downarrow; \\
\qquad\qquad [asw\uparrow \mathrm{A}\; dds\uparrow]; (ras\uparrow || cas\uparrow || dtack\uparrow || selca\downarrow); [a\uparrow \mathrm{A}\; b\uparrow] \\
\;\;||
\end{aligned}
$$

|| etc.

Figure 20: Part of the THSE specification of a DRAM controller.

4.5.3 Other synthesis results

In addition to the examples described above, another design of the SEL (SEL2) is given in the table in which the selection of the output port is performed using a single conditional signal rather than dual-rail encoding and three signal wires. The last example is the target-send burst-mode portion of a SCSI controller (TSBM) originally specified using a burst-mode finite-state machine in [34].

The results are tabulated in Table 4. First, we compared the literal counts (Lit) for the gate-level circuits derived using the generalized C-element (gC) technique and our standard C-implementation techniques. Our results show only about a 10 percent increase in literal count for generating a safe implementation that has no internal hazards. For the first three examples, the timed implementations are compared with those produced by *SYN* and SIS in terms of area represented by transistor count and delay represented as ratio of fanout of four inverter delays. The timed implementations are about 40 percent smaller and faster than the speed-independent ones produced by *SYN*. Compared with SI S, the area gains are about the same, but the improvement in delay is now about 50 percent. The table also gives the number of reachable states ($|\Phi|$) for timed and other methods showing up to two orders of magnitude less states in the timed case. In fact, due to the large state space size of the MMU example, SIS runs out of memory during synthesis. The last two examples are compared with the 3D method with the 3D specifications and results taken from [32] assuming a $0.3ns$ inverter delay in a

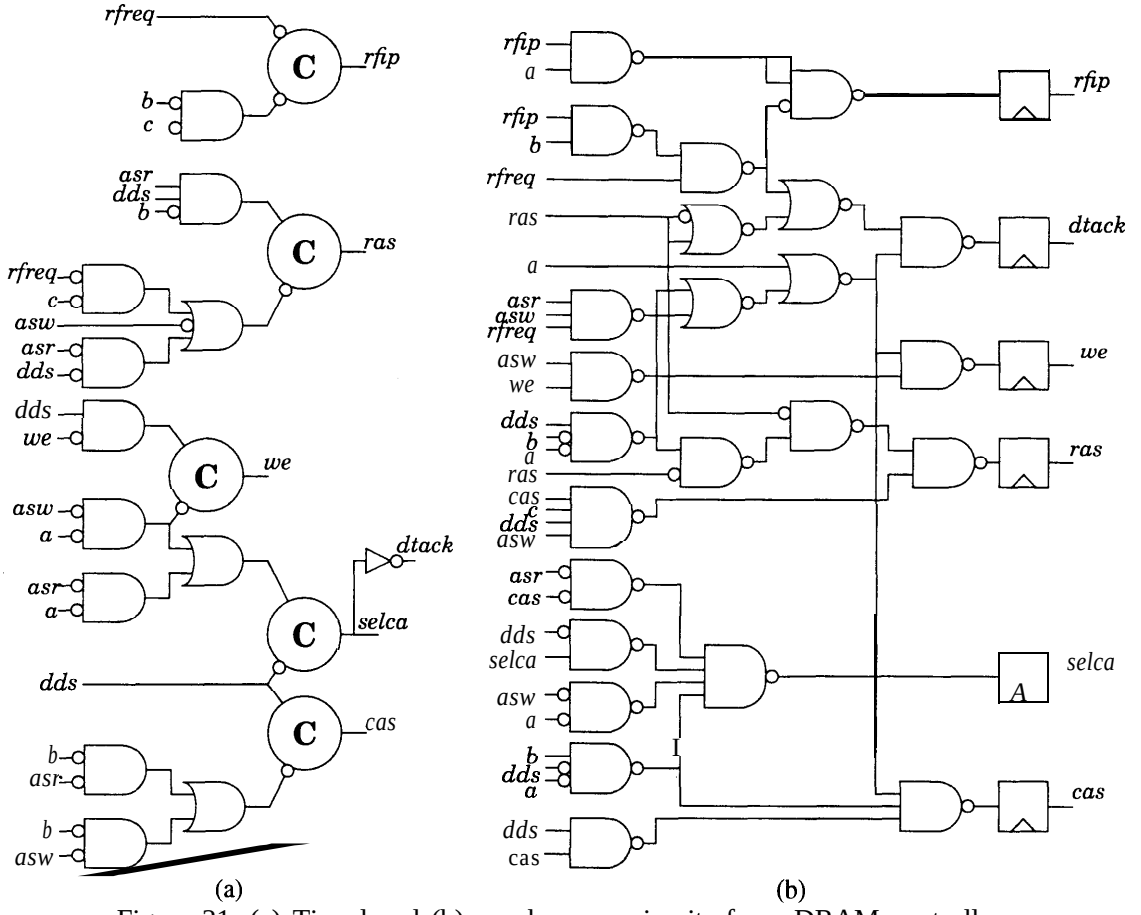


Figure 21: (a) Timed and (b) synchronous circuits for a DRAM controller.

$0.8\mu m$ CMOS process. For these designs, our timed circuits show about a 30 percent improvement in area (comparing literal count) and delay.

5 Verification procedure

Verification is the process of checking if the circuit built satisfies its specification. There are many reasons to use verification. First, even if a circuit is automatically synthesized using a formal, systematic synthesis procedure, such as ours, verification provides a double-check to discover bugs in the synthesis tools. Second, since timing assumptions must be made at the outset to synthesize a circuit, verification can be used to check these assumptions after the circuit has been synthesized. Third, designers often perform hand-optimizations to synthesized circuits, and these optimizations can be checked using verification. Finally, verification can be used to measure the robustness of the design to changes in design parameters. Although a circuit may be synthesized for one set of bounded delays, it may still work when some of the delay bounds change.

Our verification procedure requires both a specification and circuit implementation either given in or translated to an orbital net representation. The orbital net for the specification is *mirrored* (i.e., inputs and outputs are swapped) [12] and composed with the orbital net for the implementation. The state space is then explored using the partial order timing analysis algorithm described earlier. If in the process of exploring the state space a failure is detected, an error trace is returned, otherwise the timed circuit is found to implement its timed specification.

Table 4: Experimental results.

	Timed					Other Design Methodologies						
		gC	ATACS				SYN		SIS		3D	
Ex.	Φ	Lit	Lit	Area	Del	Φ	Area	Del	Area	Del	Lit	Del
SEL	53	25	27	104	5	256	160	7	158	11	n/a	n/a
SEL2	36	19	21	76	5	128	108	6.5	130	11.5	n/a	n/a
MMU	187	56	62	210	4.5	23,296	412	10	out of memory		n/a	n/a
DRAM	79	38	38	110	5.5	n/a	n/a	n/a	n/a	n/a	46	7
TSBM	113	32	33	140	4.5	n/a	n/a	n/a	n/a	n/a	58	7.5

5.1 Behavioral semantics

In order to verify our timed circuits, we adopt as our behavioral semantics trace theory as defined by Dill [12] which originated with Rem, Snepscheut, and Udding [13]. We provide structural constructions and syntactic shorthands for labeled safe Petri nets that correspond to the behavioral semantics operations. Burch [8] extended trace theory semantics to timed circuits; we extend this work with an operational formalism that allows timing in the specification, and thus hierarchical timed verification.

Dill's trace theory is based on sequences of actions, but our nets allow transitions to be labeled with sets of actions. A trace theory based on sequences of sets of actions yields a conformance relation that distinguishes, for instance, interleaved and concurrent actions. In addition, composing a net that interleaves a pair of actions with another net that has those same actions labeling one transition may lead to an unintended deadlock. We do not attempt to resolve the complexities that arise in use of such a trace theory. Instead, we define conservative structural conditions on the use of labels consisting of sets of actions that allow us to use Dill's trace theory. For instance, we cannot perform verification using traditional trace theory on the instantaneous AND function block shown in Figure 4(a). However, when we compose that model with the simple buffer given in Figure 4(b) and hide the internal wire, the resulting net contains at most a single action for each transition and traditional trace theory can be applied.

With these semantics, untimed constructions for receptiveness and synchronization apply unchanged to the timed case. Thus, implementing verification of trace structure conformance is straightforward. Determining whether an implementation conforms to a specification is reduced to determining if any of a specific set of failure transitions can be enabled. In addition, the trace theory operation of mirroring is also preserved, allowing hierarchical verification.

5.2 Generating the orbital net representations

To verify that our synthesized timed circuits implement their timed specifications, our verification procedure begins with the THSE specification and the implementation given as a netlist of basic gates. To translate the specification to an orbital net representation, the same procedure described earlier is used except that the timing requirement for each behavior place in the preset of an output transition is changed to a constraint place with timing requirement $(0, \infty\rangle c$. These constraints must be satisfied by the timed circuit implementation.

For each gate in the implementation, an orbital net is constructed corresponding to an instantaneous function block such as the one given for the AND gate in Figure 4(a). This net is composed with a delay element such as the one in Figure 4(b) with the behavioral timing requirement set by the delay given in the gate library. Each orbital net in the implementation is composed with the other orbital nets as dictated by the connections in the netlist.

5.3 Reporting failures

To determine if a timed circuit implements its timed specification, the reachable state space is found using the partial order timing algorithm for the orbital net obtained by composing the implementation with its mirrored specification. If while exploring the state space a failure is detected, a sequences of transitions found using a depth-first search is reported that demonstrates the failure. This sequence, however, may be quite long, so after reporting the failure the procedure attempts to find a shorter sequence using a breadth-first search.

5.4 Experimental results

The verification procedure described in the previous section has been automated in the tool `Orbits.` This tool has been incorporated into the design system for timed circuits `ATACS` described above. Experimental results are given in Table 5 which were run on an HP9000/735 with 144 megabytes of memory using CScheme 7.3. The left four columns indicate values that are the same for geometric and partial order timing. The startup time is the time required to parse the input and construct the appropriate orbital net. The number of net nodes is the sum of the places and transitions in the resulting orbital net. The third column gives the number of untimed states. The fourth column gives the number of discrete states, after all timing parameters are divided by their greatest common divisor. The next four columns give the number of geometric regions and the run time in seconds for verification using standard geometric timing and partial order timing, respectively.

The first half of Table 5 consists of the automatically synthesized gate-level timed circuits described above. First, we find that the number of discrete states can be quite large making discrete-time verification difficult, if not impossible. Verification of these examples using partial order timing is also more efficient than the geometric timing approach. Especially in the case of the DRAM controller where the verification time is improved by over an order of magnitude.

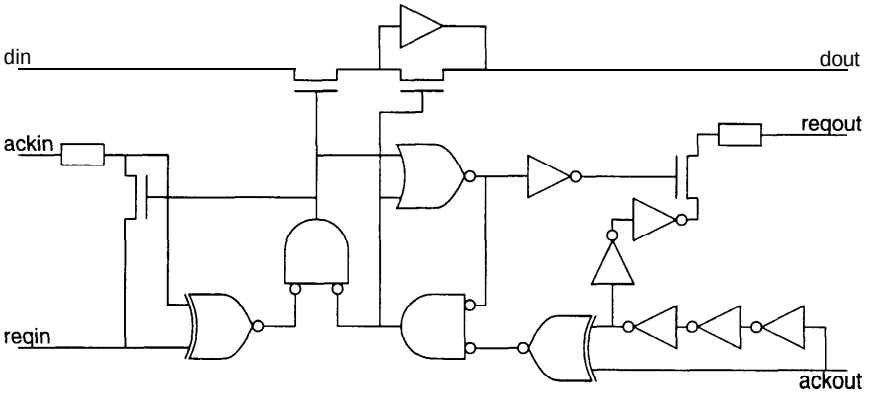


Figure 22: The Seitz queue element, a small timed circuit.

The second half of the table consists of other timed circuits and systems that exhibit a high degree of concurrency. For example, the `seit z` queue element is pictured in Figure 22; `sei tz2` is two connected copies of this circuit. The `kyy` examples [36] have thirty-seven gates and timing parameters given to three significant digits. Where the examples ran out of time or space using the geometric method, often the verification was far from done. For the `sei t z` 2 example, after one hour of CPU time, only 1,404 of the 4,572 untimed states have been seen, yet 473,202 distinct geometric regions have been encountered. One particular untimed state has 13,275 distinct geometric regions at this point. Partial order timing for this example finds the entire state space as 5,820 geometric regions in one half minute of CPU time.

Table 5: Experimental results. Time values are given in seconds. An entry of ***out of time*** indicates that the verification did not complete within two hours, and an entry of ***out of memory*** indicates that the verification ran out of memory before completing.

Examples	Startup time	Net nodes	Untimed states	Discrete states	Geometric timing regions	Geometric timing time	Partial order regions	Partial order time
SEL	2.59	770	271	6.16e5	582	1.91	358	1.76
SEL2	2.26	616	96	2033	130	0.33	102	0.29
MMU	5.94	2248	547	2.21e7	1163	5.22	583	2.03
DRAM	3.83	1326	8093	1.17e6	70611	1492.97	8899	98.13
TSBM	3.57	1464	305	49936	510	3.36	305	2.07
adv3x40	0.05	6	1	68921	1.52e5	164.99	1	0.01
adv4x40	0.03	8	1	2.83e6	out of memory		1	0.01
adv50x40	0.27	100	1	4.36e80	out of memory		1	60.21
phi13	0.19	149	144	27806	758	0.77	188	0.36
phil4	0.22	197	1152	9.82e5	out of time		1541	6.98
phil5	0.25	245	9840	3.47e7	out of time		14039	159.40
seitz	0.41	355	344	2.92e13	3234	5.48	416	1.22
seitz2	0.55	624	4572	5.48e19	out of memory		5820	29.79
kyy5	2.46	1484	5266	> le20	out of memory		6083	56.74
kyy15	1.97	1484	18357	> le20	out of memory		20250	321.47

6 Conclusion

This paper describes a methodology for the automatic synthesis and verification of gate-level timed circuits. Our synthesis procedure begins with a general THSE specification capable of specifying sequencing, concurrency, and choice. The specification is automatically translated to an orbital net representation which provides an efficient formalism for modeling timed circuit behavior. We develop a timing analysis procedure using geometric regions and partial orders to efficiently determine the reachable state space for both synthesis and verification. Using a method that decomposes the state space into regions, our synthesis procedure obtains a hazard-free timed circuit implementation using only basic gates, facilitating the use of semi-custom components. We also demonstrate the effectiveness of this synthesis procedure on several practical examples, and our results indicate that our timed circuit implementations are significantly smaller and faster than those produced by other asynchronous and synchronous design methodologies. Our verification results show that partial order timing verification can handle much larger, more concurrent examples than the standard discrete or geometric methods. Partial order timing also often finds on average very close to one and no more than two geometric regions for every untimed state which means this approach is achieving a near optimal representation of the timed state space. By applying systematic methods that incorporate timing into asynchronous circuit design, our procedure produces both efficient and reliable implementations opening the door to the use of asynchronous circuits in domains previously dominated by synchronous circuits.

Acknowledgments

We are especially thankful for invaluable comments on this work from Professor Peter Beerel of the University of Southern California. We would also like to thank Professor Steve Bums, Professor Gaetano Borriello, and

Henrik Hulgaard of the University of Washington for providing many stimulating discussions about timing analysis. We greatly appreciate the comments on this work which we received from Professor Alain Martin and his graduate students at Caltech. We would like to thank Ludmila Cherkasova and Vadim Kotov of Hewlett-Packard Laboratories for their discussions and comments during the development of `Orbits`. Finally, we are very grateful to Professor David Dill of Stanford University for his steady support and guidance.

References

[1] A. J. Martin. Programming in VLSI: from communicating processes to delay-insensitive VLSI circuits. In C.A.R. Hoare, editor, *UT Year of Programming Institute on Concurrent Programming*. Addison-Wesley, 1990.

[2] T.-A. Chu. *Synthesis of Self-Timed VLSI Circuits from Graph-theoretic Speci'cations.* PhD thesis, Massachusetts-Institute of Technology, 1987.

[3] T. H.-Y. Meng, R. W. Brodersen, and D. G. Messershmitt. Automatic synthesis of asynchronous circuits from high-level specifications. *IEEE Transactions on Computer-Aided Design,* 8(11): 1185-1205, November 1989.

[4] G. Borriello. *A New Specification Methodology and its Applications to Transducer Synthesis.* PhD thesis, University of California, Berkeley, 1988.

[5] P. Vanbekbergen. *Synthesis of Asynchronous Controllers from Graph-Theoretic Specifications*. PhD thesis, Katholieke Unviversiteit Leuven, September 1993.

[6] L. Lavagno, K. Keutzer, and A. Sangiovanni-Vincentelli. Algorithms for synthesis of hazard-free asynchronous circuits. In *Proceedings of the 28th ACM/IEEE Design Automation Conference,* 1991.

[7] C. J. Myers and T. H.-Y. Meng. Synthesis of timed asynchronous circuits. *IEEE Transactions on VLSI Systems,* l(2): 106-l 19, June 1993.

[8] J. R. Burch. *Trace Algebra for Automatic Verification of Real-Time Concurrent Systems.* PhD thesis, Carnegie Mellon University, 1992.

[9] T. G. Rokicki. *Representing and Modeling Circuits.* PhD thesis, Stanford University, 1993.

[10] S. M. Burns. Automated compilation of concurrent programs into self-timed circuits. Technical Report Caltech-CS-TR-88-2, California Institute of Technology, 1987.

[11] P. A. Beerel and T. H.-Y. Meng. Semi-modularity and testability of speed-independent circuits. *INTEGRATION, the VLSI journal,* 13(3):301–322, September 1992.

[12] D. L. Dill. Trace theory for automatic hierarchical verification of speed-independent circuits. ACM Distinguished Dissertations, 1989.

[13] M. Rem, J. L. A. van de Snepscheut, and J. T. Udding. Trace theory and the definition of hierarchical components. In R. Bryant, editor, *Third Caltech Conference on VLSI,* pages 225-239. Computer Science Press, Inc., 1983.

[14] 'R. Alur. *Techniques for Automatic Verification of Real-Time Systems".* PhD thesis, Stanford University, August 1991.

[15] T. A. Henzinger, Z. Manna, and A. Pnueli. What good are digital clocks? In *ICALP 92: Automata, Languages, and Programming,* pages 545-547. Springer-Verlag, 1992.

[16] D. L. Dill. Timing assumptions and verification of finite-state concurrent systems. In *Proceedings of the Workshop on Automatic Verification Methods for Finite-State Systems,* June 1989.

[17] H. R. Lewis. "Finite-State Analysis of Asynchronous Circuits with Bounded Temporal Uncertainty". Technical report, Harvard University, July 1989.

[18] B. Berthomieu and M. Diaz. Modeling and verification of time dependent systems using time petri nets. *IEEE Transactions on Software Engineering,* 17(3), March 1991.

[19] R. Alur, C. Courcoubetis, D. Dill, N. Halbwachs, and H. Wong-Toi. An implementation of three algorithms for timing verification based on automata emptiness. In *Proceedings of the Real-Time Systems Symposium*, pages 157-166. IEEE Computer Society Press, 1992.

[20] T. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model-checking for real-time systems. In *Proceedings of the 7th Symposium Logics in Computers Science*. IEEE Computer Society Press, 1992.

[21] N. Halbwachs. Delay analysis in synchronous programs. In Costas Courcoubetis, editor, *Computer Aided Verzjication*, pages 333-346. Springer-Verlag, 1993.

[22] T. Yoneda, A. Shibayama, B. Schlingloff, and E. M. Clarke. Efficient verification of parallel real-time systems. In Costas Courcoubetis, editor, *Computer Aided Verification*, pages 321-332. Springer-Verlag, 1993.

[23] T.-A. Chu. Synthesis of hazard-free control circuits from asynchronous finite state machine specifications. *Journal of VLSI Signal Processing*, 7(1/2):61–84, February 1994.

[24] L. Lavagno, C. Moon, R. Brayton, and A. Sangiovanni-Vincentelli. Solving the state assignment problem for signal transition graphs. *In Proc. ACM/IEEE Design Automation Conference*, pages 568-572. IEEE Computer Society Press, June 1992.

[25] P. Vanbekbergen, B. Lin, G. Goossens, and H. de Man. A generalized state assignment theory for transformations on signal transition graphs. In *Proc. International Con. Computer-Aided Design (ICCAD)*, pages 112-l 17. IEEE Computer Society Press, November 1992.

[26] R. K. Brayton, G. D. Hachtel, C. T. McMullen, and A. Sangiovanni-Vincentelli. *Logic Minimization Algorithms for VLSI Synthesis*. Kluwer Academic, 1984.

[27] P. A. Beerel, C. J. Myers, and T. H.-Y. Meng. "Automatic synthesis of gate-level speed-independent circuits", November 1994. Submitted for publication in *IEEE Transactions on Computer-Aided Design*.

[28] A. Kondratyev, M. Kishinevsky, B. Lin, P. Vanbekbergen, and A. Yakovlev. Basic gate implementation of speed-independent circuits, In *Proc. ACM/IEEE Design Automation Conference, 1994*.

[29] A. Grasselli and F. Luccio. A method for minimizing the number of internal states in incompletely specified sequential networks. *IEEE Transactions on Electronic Computers*, pages 350-359, June 1965.

[30] S. Jeong and F. Somenzi. A new algorithm for the binate covering problem and its application to the minimization of boolean relations. In *IEEE ICCAD Digest of Technical Papers*, pages 417-420, 1992.

[31] R. K. Brayton and F. Somenzi. An exact minimizer for Boolean relations. In *Proceedings IEEE 1989 ICCAD* Digest of Technical Papers, pages 316-l 9, 1989.

[32] K. Y. Yun. *Synthesis of Asynchronous Controllers for Heterogeneous Systems*. PhD thesis, Stanford University, 1994.

[33] C. J. Myers and A. J. Martin. The design of an asynchronous memory management. Technical Report CS-TR-93-30, California Institute of Technology, 1993.

[34] S. M. Nowick, K. Y. Yun, and D. L. Dill. Practical asynchronous controller design. In *International Conference on Computer Design, ICCD-1992*. IEEE Computer Society Press, 1992.

[35] E.M. Sentovich, K.J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P.R. Stephan, R.K. Brayton, and A. Sangiovanni-Vincentelli. SIS: A system for sequential circuit synthesis. Technical Report UCB/ERL M92/41, University of California, Berkeley, May 1992.

[36] K. Y. Yun. Private communication, 1993.

[37] T. G. Rokicki and C. J. Myers. Automatic verificaton of timed circuits. In *International Conference on Computer-Aided Verijcation*, pages 468480. Springer-Verlag, 1994.

[38] C. J. Myers, T. G. Rokicki, and T. H.-Y. Meng. Automatic synthesis of gate-level timed circuits with choice. to appear at 1995 Chapel Hill Conference on Advanced Research m VLSI.